NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND EVALUATION OF AN INTEGRATED, SELF-
CONTAINED GPS/INS SHALLOW-WATER AUV
NAVIGATION SYSTEM(SANS)

by

Randy G. Walker

June, 1996

Co-Advisors: Xiaoping Yun
Robert B. McGhee

Approved for public release; distribution is unlimited.

AD NUMBER		DATE	DTIC ACCESSION NOTICE
WALKER, RANDY G.		JUN 0 (1996	

1. REPORT IDENTIFYING INFORMATION

A. ORIGINATING AGENCY Superintendent
Code 82SD, Naval Postgraduate School, Mtry
CA 93943-5000

B. REPORT TITLE AND/OR NUMBER
Design and Evaluation of an Integrated,
Self-contained GPS/INSS Shallow-Water AUV

C. MONITOR REPORT NUMBER
Navigation System (SANS)

D. PREPARED UNDER CONTRACT NUMBER

2. DISTRIBUTION STATEMENT

A

DTIC FORM 50
FEB 86

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June, 1996	Master's Thesis

4. TITLE AND SUBTITLE
DESIGN AND EVALUATION OF AN INTEGRATED, SELF-CONTAINED GPS/INS SHALLOW-WATER AUV NAVIGATION SYSTEM (SANS)

5. FUNDING NUMBERS

6. AUTHOR(S)
Walker, Randy G.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
The main problem addressed by this research is to find an alternative to the use of large and/or expensive equipment required by conventional navigation systems to accurately determine the position of an Autonomous Underwater Vehicle (AUV) during all phases of an underwater search or mapping mission.

The approach taken was to advance an existing integrated navigation system prototype which combines Global Positioning System (GPS), Inertial Measurement Unit (IMU), water speed, and heading information using Kalman filtering techniques. The hardware and software architecture of the prototype system were advanced to a level such that it is completely self-contained in a relatively small, lightweight package capable of on-board processing of sensor data and outputting updated position fixes at a rate of 10 Hz; an improvement from the 5 Hz rate delivered by the prototype. The major changes to the preceding prototype implemented by this research were to install an on-board processor to locally process sensor outputs, and improve upon the analog filter and voltage regulation circuitry.

Preliminary test results indicate the newly designed SANS provides a 100% performance improvement over the previous prototype. It now delivers a 10 Hz update rate, and increased accuracy due to the improved analog filter and the higher sampling rate provided by the processor.

14. SUBJECT TERMS
Autonomous Underwater Vehicles, GPS/INS integration, navigation, NPS AUV, Kalman filtering

15. NUMBER OF PAGES
179

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

DESIGN AND EVALUATION OF AN INTEGRATED, SELF-CONTAINED GPS/ INS SHALLOW-WATER AUV NAVIGATION SYSTEM(SANS)

Randy G. Walker
Captain, United States Marine Corps
B.S.E.E, San Diego State University, 1990

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

and

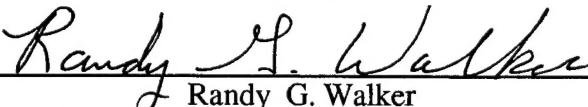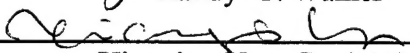## MASTER OF SCIENCE IN COMPUTER SCIENCE

from the
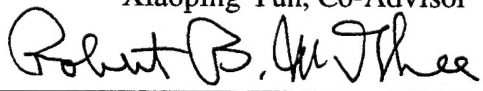
NAVAL POSTGRADUATE SCHOOL
June 1996

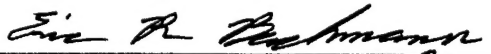Author: _____
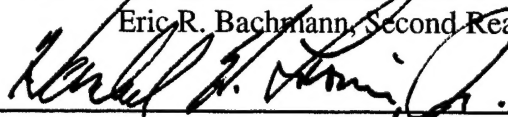Randy G. Walker

Approved By: _____
Xiaoping Yun, Co-Advisor

_____
Robert B. McGhee, Co-Advisor

_____
Eric R. Bachmann, Second Reader

_____
Herschel H. Loomis, Chairman,
Department of Electrical and Computer Engineering

_____
Ted Lewis, Chairman
Department of Computer Science

iii

ABSTRACT

The main problem addressed by this research is to find an alternative to the use of large and/or expensive equipment required by conventional navigation systems to accurately determine the position of an Autonomous Underwater Vehicle (AUV) during all phases of an underwater search or mapping mission.

The approach taken was to advance an existing integrated navigation system prototype which combines Global Positioning System (GPS), Inertial Measurement Unit (IMU), water speed, and heading information using Kalman filtering techniques. The hardware and software architecture of the prototype system were advanced to a level such that it is completely self-contained in a relatively small, lightweight package capable of on-board processing of sensor data and outputting updated position fixes at a rate of 10 Hz; an improvement from the 5 Hz rate delivered by the prototype. The major changes to the preceding prototype implemented by this research were to install an on-board processor to locally process sensor outputs, and improve upon the analog filter and voltage regulation circuitry.

Preliminary test results indicate the newly designed SANS provides a 100% performance improvement over the previous prototype. It now delivers a 10 Hz update rate, and increased accuracy due to the improved analog filter and the higher sampling rate provided by the processor.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This research was only made possible by the efforts of many people, which goes along with something I learned as a boy growing up on our family farm, "Many hands make light work." Most importantly, I would like to thank my wife Ann, and my children Christopher and Katherine. Each day they cheerfully endured the many extended periods of my absence without complaint. The work completed by this thesis would not have been as significant had I not had the peace of mind in knowing my home and family were well taken care of. Thank you Ann.

Appreciation also goes to my friend, tutor, and source of advice, Ben Salerno who continually stimulated my thinking with new perspectives on how to solve challenges encountered during this research. On many occasions, Ben displayed much patience in explaining answers to my questions only to get my "thousand yard stare" in return, and then gladly do it all over again. If it weren't for Ben's pressing and mentoring, the low-pass filter and DC-DC converter Printed Circuit Board built for this thesis, would not have come into existence.

A heartfelt thank you goes to Dr. Robert McGhee and Dr. Xiaoping Yun who served as co-advisors on this thesis. Both helped in making this work an enjoyable and rewarding experience. I particularly appreciated Dr. McGhee's patience and understanding. When things looked like they were becoming too lofty for me to accomplish, Dr McGhee would gracefully bring me back to Earth with a small dose of reality. I also particularly appreciated Dr. Yun's attention to detail and timeliness when giving me feedback on my written work.

A heartfelt expression of gratitude goes to Eric Bachmann, LT/USN, who not only was the second reader, but also assisted immensely in engineering the SANS software used in this thesis. It was surely my good fortune to have the person who authored the prototype SANS software at the software helm for my research.

A particular expression of gratitude goes to Russ Whalen. Thanks to Russ, I had a suitable, secure, and comfortable place in which to conduct my research: his office. I would also like to thank Russ for the valuable advice I received from him on all matters of thesis and life.

A special thank you goes to Dr. James Clynch, who like slashing a mighty sword through a stick of butter, quickly solved my compass checksum check problem as easily as a 10-year old solves the 2+2 problem.

DEDICATION

For Ann, Christopher, Katherine and Baby

I. INTRODUCTION

A. BACKGROUND

An Autonomous Underwater Vehicle (AUV) can be capable of numerous missions both overt and clandestine. Such vehicles have been used for inspection, mine countermeasures, survey, observation, etc. [Yuh 95]. Recent research trends in underwater robotics have emphasized minimizing the need for human interaction by increasing the autonomy of such vehicles.

The NPS 'Phoenix' AUV is an experimental vehicle designed primarily for research in support of shallow-water mine countermeasures and coastal environmental monitoring [Healey 93, 95, Brutzman 96]. In [Kwak 93], an approach is described for determining the position of submerged detected objects by executing a "pop-up" maneuver to obtain a GPS fix, and then extrapolating this fix backwards to the submerged object location using recorded inertial data. As explained in [Kwak 93], navigation accuracy during such a surfacing maneuver is strongly enhanced by the use of accurate depth information available from low-cost pressure cells. However, this form of "aided" inertial navigation [Brown 92], is not applicable to a surfaced AUV. Inertial navigation is not needed in circumstances where continuous reliable reception of GPS satellite signals is possible. However, this does not apply to AUVs, unless perhaps they are fitted with a mast to extend a GPS antenna above the effects of wave action. Such a mast is not an attractive option for military operations, and in any event may be mechanically difficult.

In efforts to overcome the problem of intermittent GPS satellite tracking for surfaced (or cruising near the surface) AUV navigation, an experimental system, using a low-cost strapped-down inertial measurement unit (IMU), has been designed to enable inertial navigation between GPS fixes. This system is well suited for pop-up navigation, so finding a means of navigating near the surface provides a complete solution to the overall navigation problem associated with transiting an AUV to a shallow water work site, recording the position of detected submerged objects, and then returning to a recovery site where stored mission data can be uploaded [McGhee 95].

Many of the missions of the Phoenix class of vehicles can be separated into two distinct phases: transit and search. After being launched from an aircraft, submarine, or surface vessel, such an AUV would conduct the transit phase of the mission in order to arrive at the search area. After

the search phase, the AUV would transit back to a recovery position. Neither of these transit phases require as high a degree of navigation accuracy as the search phase. Once established in the mission area, the Phoenix would enter the search phase which might include missions such as mine-hunting, mapping, or environmental data collection. Typically, the search phase would require more precise navigation which could be provided by more frequent GPS fixes or by using Differential GPS (DGPS) or post-processing, if available. Both mission phases may involve waypoint steering and obstacle avoidance.

One of the most important and difficult aspects of an AUV mission is navigation. It is important that the navigation system be robust if the AUV is to be capable of a wide variety of missions. In order to achieve such robustness, the AUV should be capable of navigating with the Global Positioning System (GPS) and/or an Inertial Navigation System (INS). The GPS is capable of highly accurate positioning when the AUV is surfaced, while an INS can be used for submerged navigation. In order to ensure accurate navigation for the various missions, the GPS and INS components can be combined. A favorable analysis of this type of navigation system was conducted in [McKeon 92]. The hardware and software architecture required for a typical mapping scenario was evaluated in [Norton 94].

[Bachmann 95] made the architecture evaluated in [Norton 94] a reality, and subsequently developed the first working prototype of the proposed Shallow-water AUV Navigation System (SANS). With the prototype SANS having achieved favorable results in open-water, at-sea test trials, the research reported in this thesis advances the SANS to another level of maturity, making it now ready for direct application to a real-world AUV.

B. RESEARCH QUESTIONS

This thesis will examine the following research topics:

- Evaluate the hardware and software architecture of the GPS/INS prototype SANS.

- Evaluate the feasibility of an AUV accurately navigating from point to point using GPS/INS while conducting open-ocean transit.

- Develop a hardware configuration which will enable the GPS/INS SANS to be housed in one small, self-contained package.

C. SCOPE, LIMITATIONS AND ASSUMPTIONS

This thesis reports the findings of the fifth year of research in an ongoing research project. The scope of this thesis is to investigate the feasibility of an AUV navigating from point to point using a combination of GPS/INS. The requirements for a SANS described by [Kwak 93] which impact this project are:

- Low power consumption. Operation from an appropriately sized external battery pack for 12 hours is desirable.

- Limited exposure time. The amount of time that the GPS antenna is exposed in the search phase should be as short as possible. Up to 30 seconds of exposure is allowed, but time between exposures should be maximized.

- Maintain clandestine operation. The GPS antenna should present a very small cross section when exposed and should not extend more than a few inches above the surface of the water.

- Maximize accuracy. During the search phase of the mission, system accuracy of 10 meters or better is required with postprocessing, both submerged and surfaced.

- Total volume not to exceed 120 cubic inches. Elongated, streamlined packaging is preferred.

For the purposes of this research, DGPS will be used as ground truth data (without postprocessing) for determining appropriate Kalman filter gains. However, some real-world scenarios will only utilize the noncorrected GPS signal for real-time mission navigation and may require further tuning of Kalman filter coefficients.

D. ORGANIZATION OF THESIS

The purpose of this thesis is to present the development of a system meeting the mission requirements of the SANS. The term AUV is understood to include any small underwater vehicle (including human divers) which can easily carry such a compact device. The term "towfish" refers to the test vehicle (depicted in Figure 2.2) used to evaluate the SANS during at-sea testing.

Chapter II reviews previous work on this project as well as previous work on GPS navigation and AUV submerged navigation.

Chapter III provides an evaluation of the prototype SANS in the form of both a time domain and frequency domain analysis of the IMU sensor output signals.

Chapter IV is a detailed description of the hardware currently in use for this portion of the project.

Chapter V is a detailed description of the software changes, additions, and updates made to support this portion of the project. The chapter includes a description of the C++ code required for future towfish experiments. This description provides an explanation of the class and object hierarchy used, as well as an outline of the major functions that were added as a result of this research.

Chapter VI is a description of the experiment design and analysis of the experimental results.

Chapter VII presents the thesis conclusions and provides recommendations for future research.

II. SURVEY OF RELATED WORK

A. INTRODUCTION

Autonomous Underwater Vehicles (AUVs) have the potential to be used in an efficient and cost effective manner in a variety of missions involving military and non-military applications. One of the most important aspects of an AUV mission is the ability to navigate accurately. Many possible missions, such as mine-hunting, require a high degree of navigation accuracy. This chapter will discuss some of the solutions for navigating an AUV.

In general there are two categories of navigation systems: those that are based on external signals and those that are based on sensors. External-signal-based navigation systems such as, Loran, Omega, and GPS are only able to determine position while the signal receiver is exposed to the signal. Loran and Omega are relatively inaccurate compared to GPS. While Loran covers almost the entire northern hemisphere, it has almost no coverage in the southern hemisphere [Bowditch 84]. GPS on the other hand is capable of world-wide coverage with a high degree of navigational accuracy.[Logsdon 92]

Sensor-based navigation can be implemented as a self-contained unit which can be composed of various types of equipment such as IMUs, acoustic transponders, or geophysical map comparison. Each of these components has its disadvantages. Acoustic transponders must be pre-deployed at precisely known locations and may require costly maintenance. Geophysical map interrogation requires a precise bottom contour map previously stored in the AUV's computer. IMU-based navigation is prone to sensor drift, which if left uncorrected, can become very large.

B. GPS NAVIGATION

The Navigation Satellite Timing and Ranging (NAVSTAR) Global Positioning System (GPS) is a space-based radio positioning, navigation and time-transfer system sponsored by the U.S. Department of Defense (DoD). It was originally intended to provide the military with precise navigation and timing capabilities [Parkinson 80]. The system is designed to provide 24-hour, all-weather navigation by providing total earth coverage using 24 satellites in 22,200 km orbits that are inclined 55°, with 12 hour periods. The satellites broadcast two L-band frequencies: L1 (1575.4 MHz) and L2 (1227.6 MHz). Navigation and system data, predicted satellite position (ephemeris) information, atmospheric propagation correction data, satellite clock error informa-

5

tion, and satellite health data are all superimposed on these two carrier frequencies [Logsdon 92, Wooden 85].

There are two different navigation services available from the GPS satellites depending on the type of receiver being used: the Standard Positioning Service (SPS) and the Precise Positioning Service (PPS). The SPS is achieved by receiving the L1 carrier signal which is broadcast with an intentional inaccuracy called Selective Availability (SA). SA limits world-wide navigation to 100 m horizontal accuracy with a 95% confidence level [Logsdon 92]. The PPS is limited to U.S. and allied military, and specific non-military uses that are in the national interest. Access to PPS is restricted by use of special cryptographic equipment. PPS provides the highest stand alone accuracy: 15 m Spherical Error Probable (SEP), a velocity accuracy of 0.1 m/sec, and a timing accuracy of better than 100 nanoseconds [Logsdon 92, Wooden 85].

In order to take full advantage of GPS precision without having access to cryptographic equipment, civilian customers have determined a way to improve the accuracy of the SPS. The most common way to work around the inaccuracies of the SPS is Differential GPS (DGPS), which may be used in real-time or during post-processing. DGPS is a method which allows highly accurate information to be obtained from GPS without the cryptographic equipment required for access to the P-code of PPS. The idea behind DGPS is to survey a receiver at a stationary site, allow the stationary site to determine the difference between its actual position and its GPS position, and broadcast the pseudorange corrections to any DGPS capable receivers. Real-time differential processing can reduce the typical 100 m accuracy of the SPS to 2-4 m regardless of the status of SA [Logsdon 92]. In the case of post-processing, it is possible to have the AUV record the raw PPS or SPS GPS information for later comparison to a known geographical site. Precise post-processing procedures can be used to reconstruct extremely accurate positioning information, typically in the submeter range. Table 1 shows a comparison of expected GPS accuracies.

POSITIONING SERVICE	PPS (m)	SPS (m)
Non-Differential	16	100
Differential	2-4	2-4

Table 1: Expected RMS GPS accuracy levels [Logsdon 92]

As GPS technology has matured, the size and cost of GPS receivers has decreased drastically. Not only is miniaturization improving, but GPS receivers have maintained or increased in performance capability. Since as early as 1992, the GPS industry has been able to produce receivers that are essentially a single printed circuit board. [Souen 92] reports that the Furuno GPS receiver module LGN-72 is an eight-channel receiver implemented on a single printed circuit board measuring 100 mm x 70 mm x 20 mm and requiring only 2 W of power.

There is currently however, a performance trade-off associated with the miniaturization of GPS receivers. Trimble currently offers the PC Card 110 GPS receiver in the form of a Personal Memory Card International (PCMCIA) interface. This credit card sized device simply slides into any laptop, most palmtops, or pen-based computer compliant with PCMCIA (release 2.0). This miniature GPS receiver is capable of tracking eight satellites using three channels. However, because it does not have an allocated channel for each of the eight satellites it's capable of tracking, it does not use a continuous tracking scheme, which degrades its acquisition time performance. In order to reduce the size of the receiver, manufacturers are reducing the number of channels on the receiver. In this configuration, GPS receivers are called "sequencing" receivers [Logsdon 92]. Sequencing receivers utilize a time-sharing technique to "dwell" on each satellite for a brief interval before switching to the next satellite in the sequence. Sequencing receivers have a typical acquisition time of about 2 minutes. Continuous tracking GPS receivers have a typical acquisition time of about 30 seconds, however, they are less compact in size as they have more receiver channels. Given this trade-off relationship between size and performance, the choice of GPS receiver must be made with the particular application in mind. If the application is not so dynamic (i.e., mobile navigation), a sequencing receiver would offer an adequate compromise. However, if the application requires a short time to initial acquisition, the most viable option is the continuous tracking receiver.

Given the level of miniaturization and performance along with its excellent accuracy, GPS is an obvious choice for AUV navigation. One manner of using GPS to locate an AUV is to place buoys with GPS receivers at appropriate locations. These buoys would translate the GPS signal and retransmit an underwater acoustic signal. The AUV would determine its position via ranging and position fixes to the buoys. [Youngberg 91] suggests that the GPS antenna, receiver, processing and control subsystem, acoustic transmitter, battery power, and homing beacon could all be contained in a buoy measuring 123 mm diameter x 910 mm long and weighing 5-15 kg. A

simulation which showed the feasibility of this approach is presented in [Leu 93]. The simulation consisted of several sonobuoys spaced one kilometer apart. Due to uncertainties in buoy position caused by wave action and variations in altitude, the study proposed the use of Kalman filtering techniques to combine the outputs of an accelerometer and DGPS to enhance accuracy. Each GPS buoy would essentially act as a GPS satellite and broadcast its position via spread spectrum signals used by the AUV for ranging. This technique would eliminate the requirement to predeploy a surveyed transponder field.

Another possible method for using GPS to determine the AUV's position is to physically mount the GPS antenna and receiver onboard the AUV. One major concern would be that the GPS receiver would be unable to acquire satellites in a timely manner suitable to the mission due to splash effects on the antenna. [Norton 94] describes both static and dynamic test results which show that a submersible system is able to meet the accuracy and time requirements of the mission while being splashed by wave wash.

C. INERTIAL NAVIGATION

Inertial navigation is basically a complex method of dead reckoning. In its purest form it involves no outside references to fix position. All position data is calculated relative to a known starting point. An inertial navigation system (INS) continuously measures three mutually orthogonal acceleration components using accelerometers. These measurements are taken in short time increments and multiplied by elapsed time in order to determine an estimate of instantaneous velocity. The three-dimensional change in position can then be determined by integrating respective velocities over time. [Bachmann 95]

The primary drawback of any INS is the tendency for small sensor drift rates to accumulate errors over time. Without outside references for correction, these errors grow relentlessly and eventually lead to large errors in the estimated position. Highly accurate inertial navigation systems can be constructed, but they are large, costly, and complex [Tuohy 93]. Size alone makes them unacceptable candidates for the SANS. In order to meet the SANS requirements, a low-cost INS can be integrated with GPS. GPS will provide the INS with the periodic position fixes necessary to correct slowly building INS errors.

The acceleration measurements required by an INS can be made by several types of IMUs. These can be divided into two fundamental categories: gimbaled and strapdown. Due to their large

size and power requirements, gimbaled systems are not suitable for the SANS. In a strapdown unit, three mutually orthogonal accelerometers and three angular rate sensors are mounted parallel to the three body axes of the vehicle. Changes in linear and rotational velocities are continuously measured. Strapdown systems are smaller and simpler than gimbaled systems, but necessitate much larger computational capabilities [Logsdon 92].

D. INTEGRATED GPS/INS NAVIGATION

SPS could be used to adequately perform both the transit and search phases of an AUV mission. During the transit phases, non-differential SPS and a magnetic compass would provide the primary source of navigation data. In order to utilize GPS as a meaningful correction to a low-cost INS system, periods between fixes during the transit phase must not exceed the time in which an AUV could travel a distance greater than the horizontal accuracy of SPS (100m). The mapping phases of an AUV mission would require the vehicle to maintain more accurate navigational picture both submerged and on the surface. This would necessitate the use of periodic differentially corrected GPS information in order to keep the INS system accurate while submerged. This differential correction could be provided in real-time during overt missions along friendly shores, provided a DGPS reference signal is available, or during mission post-processing following a clandestine mission.

Integration of GPS and INS into a single system can produce continuously accurate navigational information even when using relatively low-cost components. This integration not only allows periodic reinstallation of the INS to correct accumulated errors but can also (with the aid of Kalman filtering techniques) improve the performance of the INS between fixes. Filtering the acceleration data with additional sensor information such as water speed and heading will further improve the quality of the integrated system. Overall, an integrated system will provide improved reliability, smaller navigation errors and superior survivability [Logsdon 92].

Kalman filtering is a method of combining all available sensor data regardless of their precision to estimate the current posture of a vehicle [Cox 90]. The filter is actually a data-processing algorithm which minimizes the error of this estimate statistically using currently available sensor data and prior knowledge of system characteristics. Each piece of data is weighted based upon the expected accuracy of the measurement it represents. In a complementary filter, low-frequency data, which is trusted over the long term, and high-frequency data, which is trusted

only in the short term, are used to "complement" each other providing a much better estimate than either can alone [Brown 92].

[Bachmann 95] demonstrated the use of this complementary filter technique by combining the low-frequency data of the accelerometers and compass with high-frequency angular rate and heading information. Intermediate position results were obtained by integrating high-frequency water-speed data. GPS data was used to reinitialize the system each time a fix was obtained and develop an error bias, expressed as an apparent current, to correct the system between fixes. The concept of using the relatively inexpensive IMU with limited accuracy coupled with differentially-corrected GPS has proven to be a viable solution to the challenge of shallow-water AUV navigation [Bachmann 95].

E. AUV SUBMERGED NAVIGATION

There are many techniques available for submerged navigation, including dead reckoning, inertial, electromagnetic and acoustic navigation. With acoustic navigation, time of arrival and direction of propagation of acoustic waves are the two principal measurements made. A wide variety of acoustic navigation systems have been developed for underwater vehicle use. They are typically divided into long, short, and ultrashort baseline systems (LBL, SBL, and USBL). All involve the use of an array of acoustic beacons or receivers whose positions must be known to an accuracy better than the desired vehicle localization accuracy [Tuohy 93]. Unfortunately, most acoustic navigation systems require major expeditions for their accurate set-up and periodic main-tenance. This makes them expensive and in many ways reduces the level of autonomy achievable by an AUV. Also, acoustic navigation methods are affected by changes in the speed of sound in the ocean and suffer from refraction and multipath propagation problems in restricted shallow water coastal and ice-covered areas [Tuohy 93].

There are various other ways of determining a vehicle's velocity and position while submerged without the aid of external signals. An AUV could use Doppler sonar to determine velocity. Charge Coupled Device (CCD) cameras, laser scanning, or variations in the earth's magnetic field can also aid in determining position [Bergem 93]. Position could also be estimated by the double integration of acceleration as sensed by an Inertial Measurement Unit (IMU).

Unless an AUV has access to outside references, it will not be able to refer to external signals while submerged. If this is the case, then the system must rely on some sort of dead reckoning.

Modern dead reckoning systems typically use magnetic or gyroscopic heading sensors and a bottom or water-locked velocity sensor [Grose 92]. The main problem is that the presence of an ocean current will add a velocity component to the vehicle which is not detected by the speed log. In the vicinity of the shore, ocean currents can exceed two knots [Tuohy 93]. Using dead reckoning with currents which are relatively large in relation to the typical 4-6 knot speed of an AUV can produce extremely inaccurate results [Tuohy 93].

There are many techniques for measuring accelerations and angular rates. These include using ring laser and fiber optic gyros, rotating mass gyros, vibratory rate sensors, and high performance IMUs. Inertial grade IMUs typically contain three angular rate sensors, three precision linear accelerometers and a three-axis magnetometer. The acceleration measurements required by an Inertial Navigation System (INS) can be made by several types of IMUs. Again, these can be divided into two fundamental categories: gimbaled and strapdown. All of these sensors are subject to drift errors which relentlessly increase with time. High quality sensors are subject to less drift but can cost up to $100,000 [Tuohy 93], making them unattractive for small AUVs.

[McKeon 92] proposes a combination of GPS and INS to allow an AUV to determine position information. While submerged, the AUV uses a low-cost inertial navigation system. However, when on the surface the vehicle has access to GPS information. GPS/INS information could be combined with a Kalman filter techniques to reduce the errors during the next dive sequence as simulated in [Nagengast 92] and demonstrated in [McGhee 95].

F. THE PREVIOUS PROTOTYPE

[Bachmann 95] describes in detail the previous hardware and software configurations of the SANS. For the benefit of the reader, Figures 2.1, 2.2, and 2.3 are given again in this thesis to aid in discussing the previous prototype. Figure 2.1 shows a block diagram of the hardware assembled for at-sea testing of the SANS system. As seen in Figure 2.1, the system relied on an external 386 computer for processing sensor data, a modem connection for transmitting data packets to the towing vessel, and a coax cable to receive GPS data from the towed vehicle GPS antenna. The output of the A/D converter was being fed to the data-logging computer via an RS-232 connection, and the GPS and DGPS receivers were physically located on the towing vessel. The hardware configuration for this research has been changed considerably, and will be presented in a subsequent chapter.
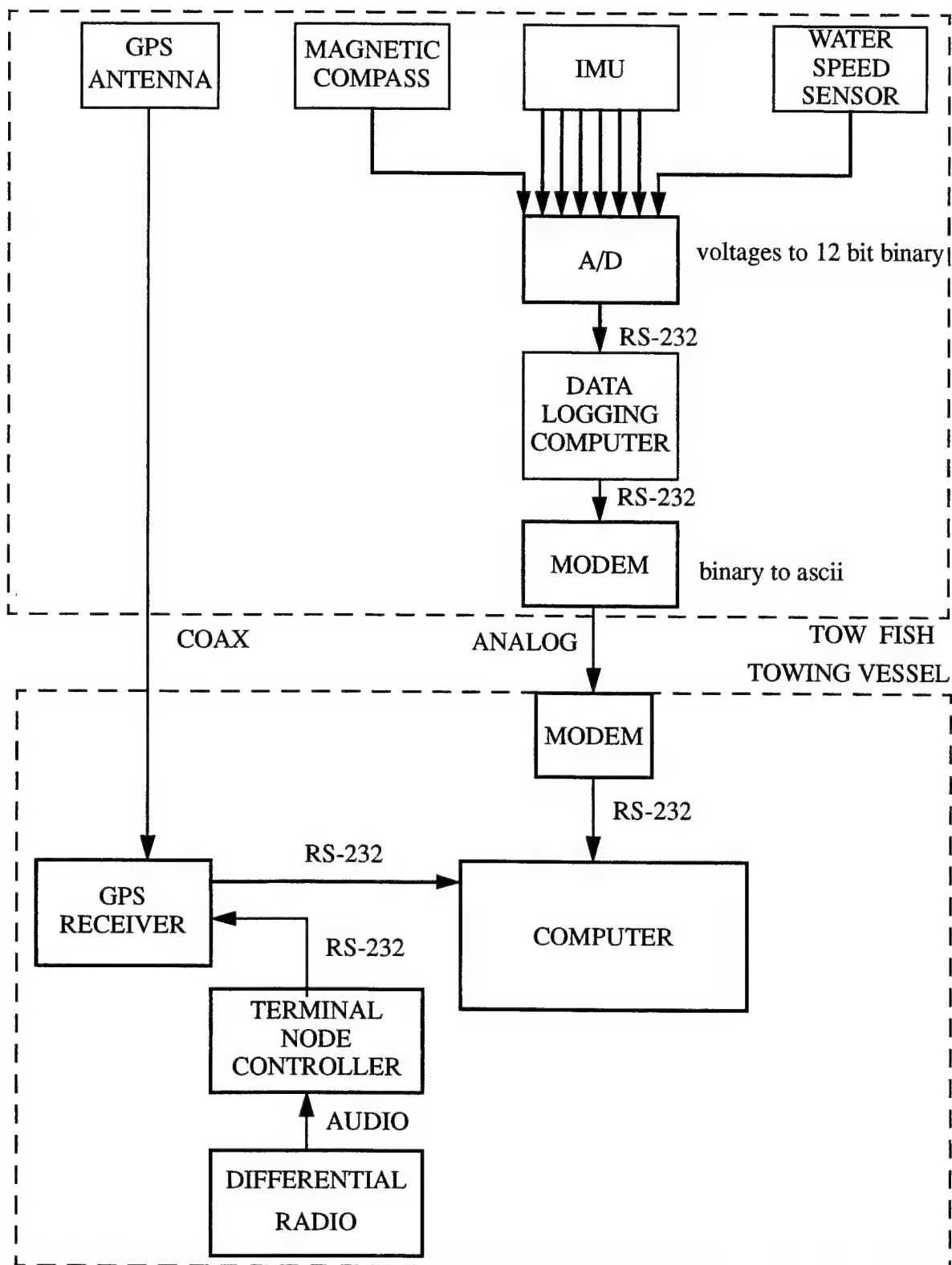
Figure 2.1: Towfish Experiment Hardware Configuration
[Bachmann 95]
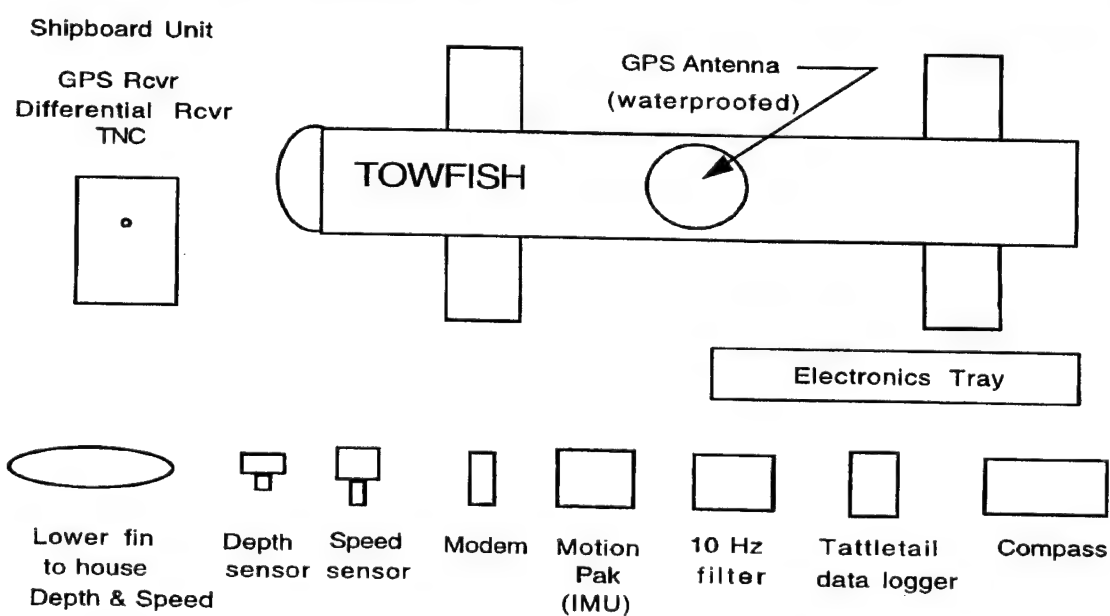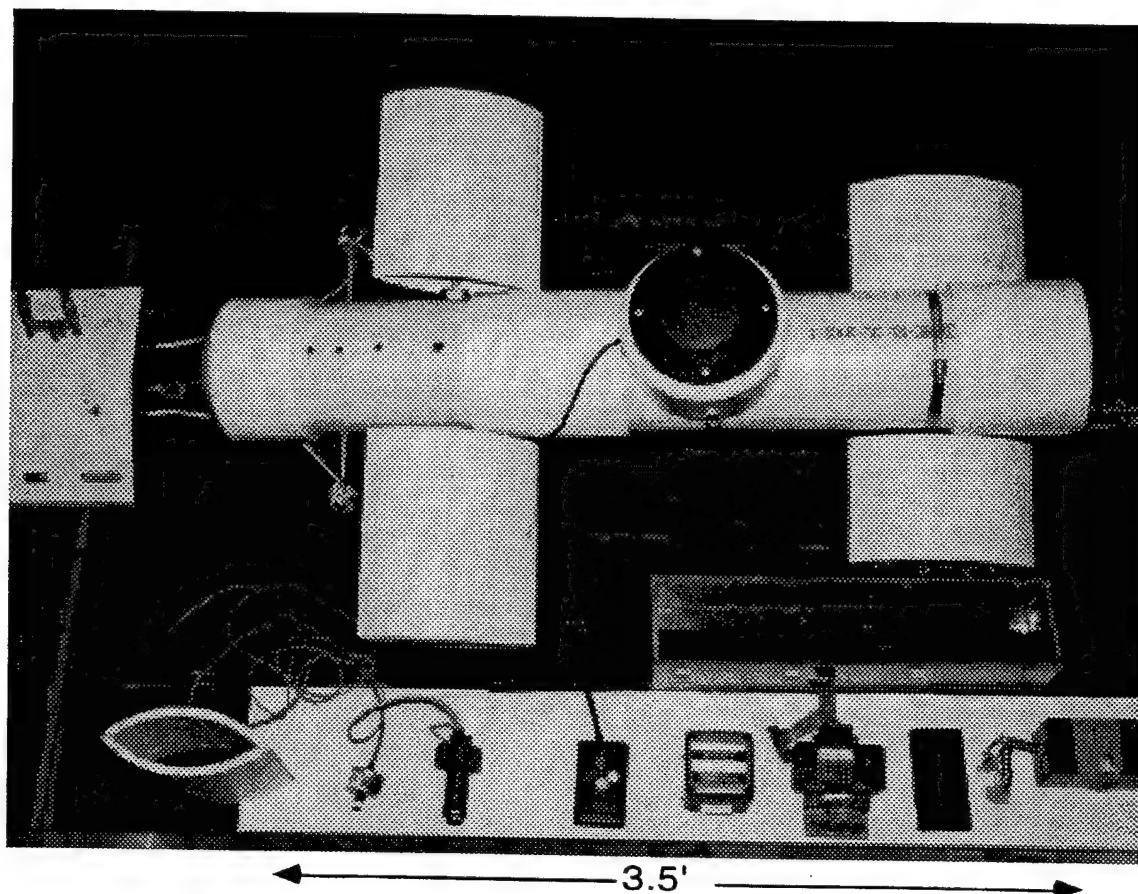
Figure 2.2: SANS and Towfish Components [Bachmann 95]
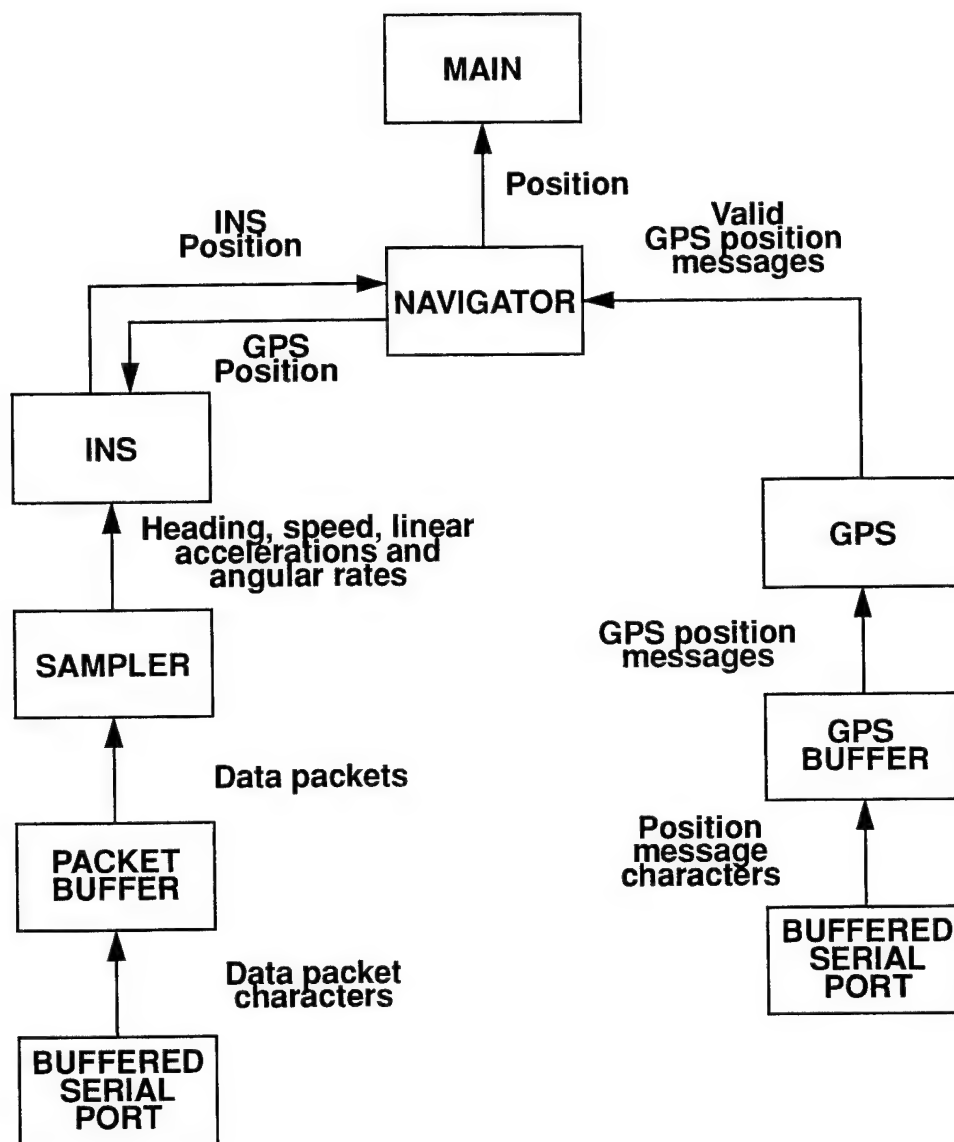
Figure 2.3: SANS Software Objects and Data Flow [Bachmann 95]

Figure 2.2 presents a photograph of the major components of the previous prototype. Figure 2.3 shows the SANS (as previously configured) software objects and the types of data passed from one to another. In its current configuration, the INS object no longer receives data from the sampler object, but rather from processor registers. Again, changes to the SANS software configuration will be presented in more detail in a subsequent chapter of this thesis.

G. SUMMARY

The above survey has shown that there are many ways to overcome the challenges associated with AUV navigation. The choices range from simple dead reckoning to systems which use acoustic information from floating or stationary transponders, to complex systems which use sophisticated IMUs and GPS receivers combined with Kalman filtering techniques.

Many AUV missions could be accomplished using an integrated navigation system combining GPS and INS. Similar systems in other applications have been demonstrated to have superior GPS signal acquisition and reacquisition performance whenever loss of lock occurs. This results in improved survivability in hostile environments and smaller navigation errors. This research continues an ongoing experimental study pertaining to the development of such a system and the associated problems. The current system under evaluation is of small physical size and relatively low cost. The IMU selected is representative and has limited accuracy, so additional water-speed and magnetic heading information is required. This means that accelerometers are used mainly to derive low frequency attitude information, and are not utilized for velocity or position estimation over long periods.

The availability of differential GPS in open-ocean tests in Monterey Bay will allow the experimental choice of navigation filter gains to accurately assess overall system performance in a variety of sea states and for various operational scenarios. Previous research on the prototype SANS has produced test results and qualitative error estimates which indicate that submerged navigation accuracy comparable to GPS surface navigation is attainable. The research goal of this thesis is to refine the error estimates and the hardware configuration to allow more prolonged submerged navigation, and develop the SANS into a self contained system capable of being internally or externally attached to any AUV and delivering regular, accurate position updates.

III. EVALUATION OF THE PROTOTYPE SANS HARDWARE

A. INTRODUCTION

At the heart of the prototype SANS are the GPS receiver, the IMU, and other sensor devices. The GPS receiver, compass, and water speed sensor have not, up to this point in the course of this research, presented any serious problems in accuracy or dependability. However, the IMU outputs, after being fed through signal processing and conditioning circuitry, are suspected sources of navigation inaccuracies [Bachmann 95].

This chapter does not provide any evaluation of the GPS/DGPS receivers, water speed sensor, or compass since there is no apparent major error contributions from these devices. It does, however, provide an investigation into what noise is present in the real-world Systron-Donner MotionPak IMU. It also provides an evaluation of the low-pass filtering and conditioning circuitry. Finally, it presents an evaluation summary of the prototype SANS hardware.

B. NOISE CHARACTERISTICS OF THE MOTIONPAK INERTIAL SENSORS UNIT

The Systron-Donner Model MP-GCCCQAAB-100 "MotionPak" inertial sensor unit consists of a cluster of three accelerometers and three "Gyrochip" angular rate sensors. The accelerometer specifications are shown in Table 3.1. The angular-rate sensor specifications are given in Table 3.2.

With the IMU placed on a stable test bench, polariod photographs were taken of an oscilloscope screen display of each sensor output. Figure 3.1 depicts the respective noise characteristics in each of the IMU sensor outputs. This analysis gives hints there is broadband, "white" noise in the accelerometer sensors, while there is a 275 Hz sinusoidal signal present in the angular-rate sensors. Figures 3.2 and 3.3 depict the results of a power spectrum analysis of the accelerometer and angular-rate sensor outputs. Figure 3.2 shows the power spectrum between 8 Hz and 100 KHz as well as between 8 Hz and 50 Hz. The spectrum analyzer used for these tests has its own power spectrum which was sufficiently below the sensor signals by 8 Hz. Thus 8 Hz was chosen as the start frequency. As these plots show, with a reference level of -20 dB (the reference level is depicted as the top-most horizontal index line), and using the value of 10 db/Div, the signal has a maximum value of -60 dB below 50 Hz. The signal energy drops off above this point and becomes relatively flat in the vicinity of 100 KHz. Figure 3.2 shows that the noise in the acceler-

Parameter	Unit	x-axis	y-axis	z-axis
Range	g	1	1	2
Scale Factor	V/g	7.469	7.478	3.727
Scale Factor Temp. Coefficient	%/deg C	0.001	-0.002	-0.001
Bias	mg	-2.447	4.570	-0.586
Bias Temp. Coefficient	μ g/deg C	-47	-66	-48
Sensitivity	μ g	10	10	10
Bandwidth	Hz	797	757	901
Output Impedance	Ohms	2464	2494	1177

Table 3.1: MotionPak Accelerometer Sensor Specifications [Systron-Donner 94]

Parameter	Unit	x-axis	y-axis	z-axis
Range	deg/sec	50	50	50
Scale Factor	mV/deg/sec	50.151	49.906	50.242
Scale Factor Temp. Coefficient	%/deg C	0.03	0.03	0.03
Bias	deg/sec	-0.06	0.23	0.23
Bias Temp. Coefficient	deg/sec P-P	3	3	3
Sensitivity	deg/sec	0.002	0.002	0.002
Alignment	degrees	0.26	0.41	0.34
Noise	deg/sec/$\sqrt{Hz}$	0.008	0.009	0.008
Bandwidth (to -90 deg phase)	Hz	70	71	71

Table 3.2: MotionPak Angular-Rate Sensor Specifications [Systron-Donner 94]
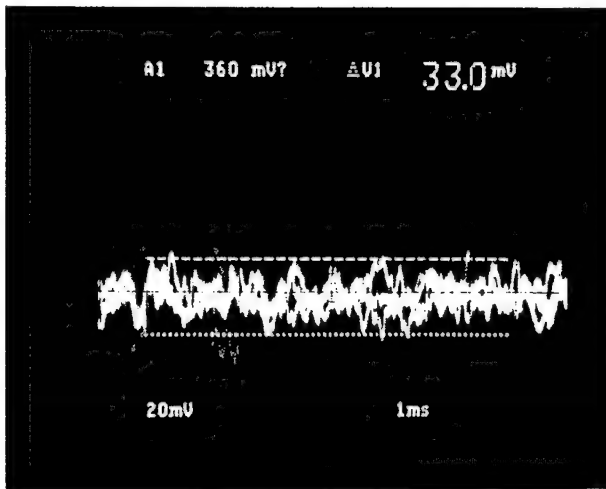
X-accelerometer
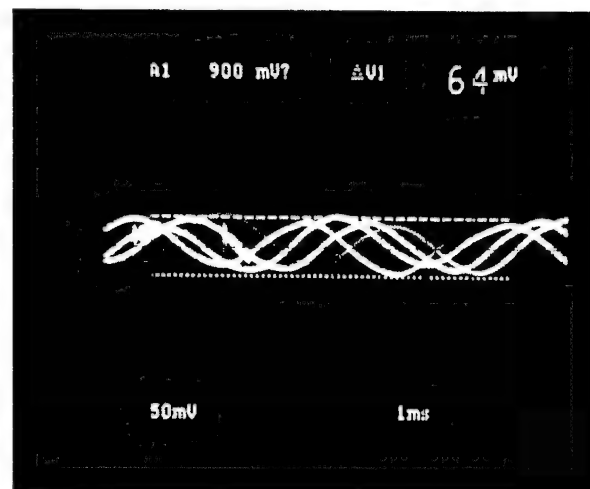
X-angular-rate
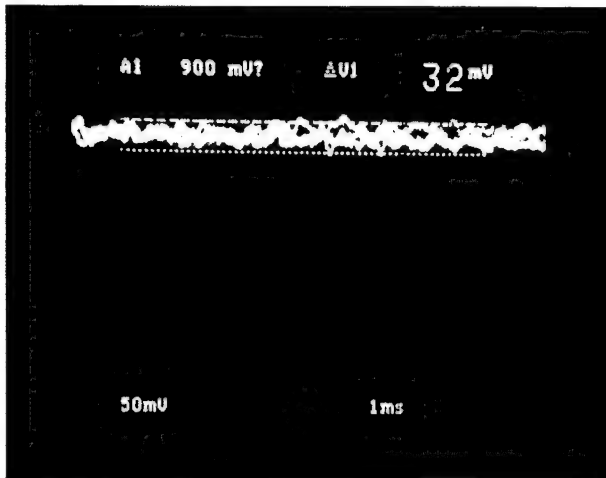
Y-accelerometer
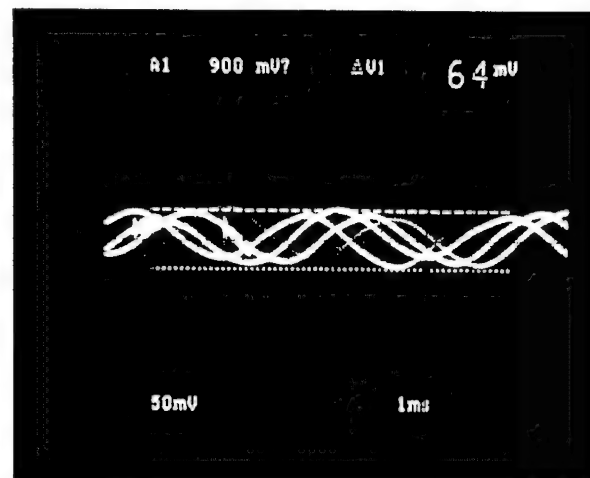
Y-angular-rate
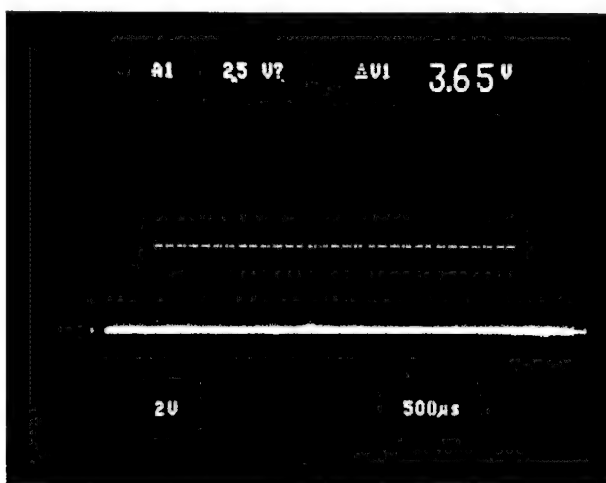
Z-accelerometer
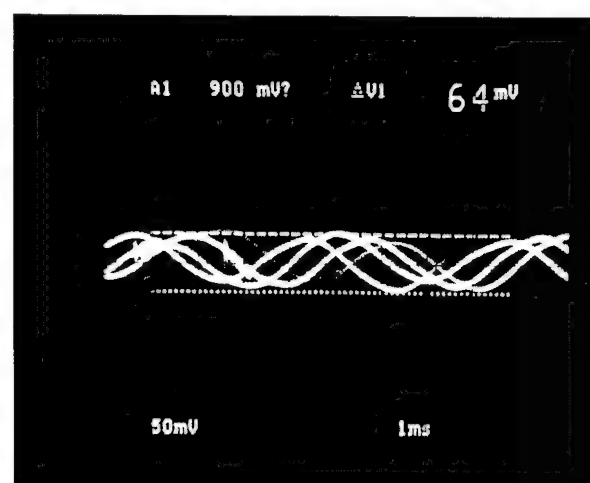
Z- angular-rate

Figure 3.1: Systron-Donner IMU Sensor Noise Characteristics

19

ometers is in fact broadband white noise, since it shows a power spectrum with energy across a wide range of frequencies. The white noise is most likely thermal and low-level EMI noise. Figure 3.3 shows there is more "color" in the angular-rate sensor noise. Mainly, there appears to be harmonic noise between 1.5 KHz and 300 KHz and several high-power noise peaks at 100, 200, and 275 Hz. The peak at 275 Hz is most likely harmonic noise from the fundamental tuning-fork in the rate sensors. [Matthews 95] describes an investigation of signal noise in this same Systron-Donner MotionPak IMU. [Matthews 95] found a strong peak at 275 Hz, and attributed this noise to the internal tuning-fork oscillator in the sensor itself.

The accelerometer sensor outputs depicted in Figure 3.1 show a DC bias of -21 mV and 125 mV respectively. These DC biases are most likely caused by a combination of electronic sensor bias and slight tilts in the test bench surface. For the purpose of this evaluation, these DC biases are ignored. Furthermore, the software as described in [Bachmann 95], uses bias correction factors that essentially negate any sensor bias affects on navigation accuracy.
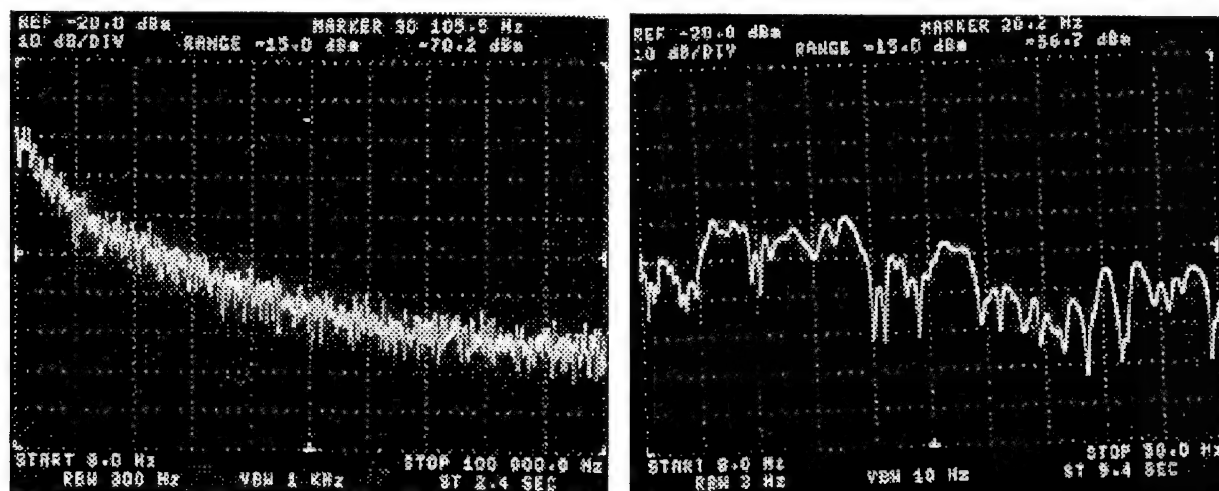
Figure 3.2: Power Spectrum of Accelerometer Sensors
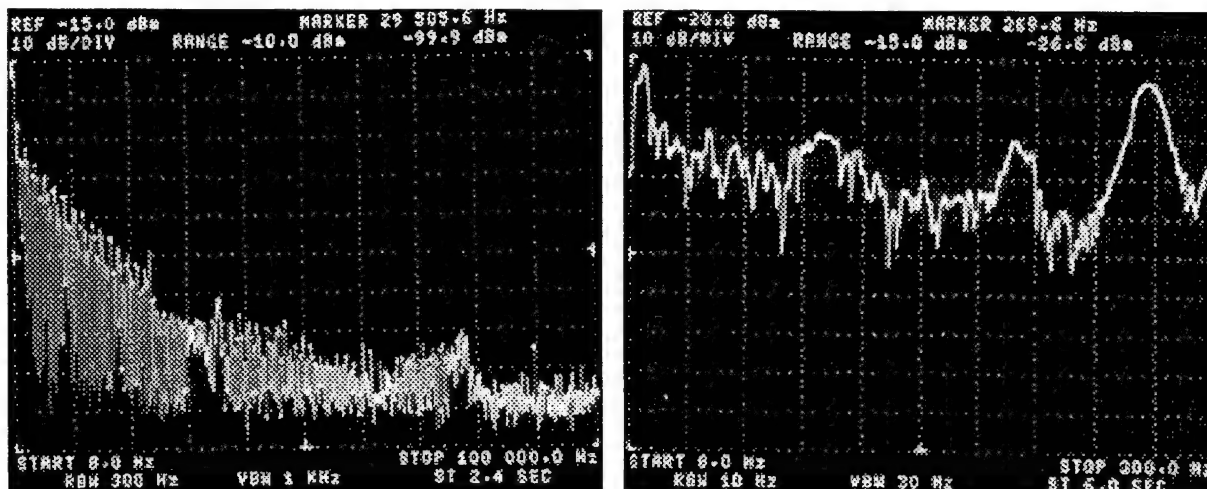
Figure 3.3: Power Spectrum of Angular-rate Sensors

C. THE SIGNAL PROCCESSING AND CONDITIONING CIRCUITRY

The signal processing and conditioning circuitry, described in detail in [Schubert 95], low-pass filters the IMU outputs at a cutoff frequency of 10 Hz using an active, anti-aliasing, two-pole Sallen-Key Bessel filter design. This circuit further converts the dual-ended output swing of the IMU to a single-ended 0-5 volts.

Figure 3.4 depicts the frequency response of the low-pass filter. Generally, this analysis confirms that the -3dB frequency is about where it's supposed to be, and the response does in fact rolloff at the expected -40dB/decade.
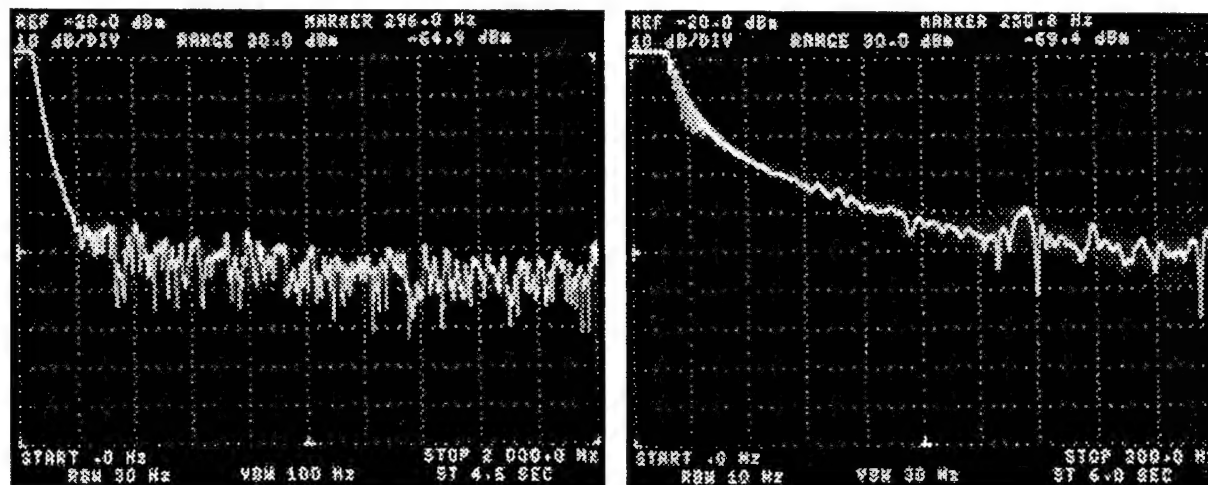
Figure 3.4: Frequency Response of the Low-pass Filter

21

For the SANS application, in order for any sensor noise to not degrade the accuracy or resolution of the measured signal, the energy in the filtered signal above 10 Hz must be attenuated down below:

$$20\log\frac{1}{2^{12}} = -72dB$$

As seen in Figure 3.4, the low-pass filter is not successfully able to attenuate signal noise energy in the stop-band below approximately 1 KHz (the center horizontal index line is at -70 dB). The results of this filter short-fall can be seen by looking at the power spectrum of the accelerometer and angular-rate sensor outputs after low-pass filtering. Figure 3.5 and Figure 3.6 depict the power spectrum of the filtered accelerometer and angular-rate sensors respectively.
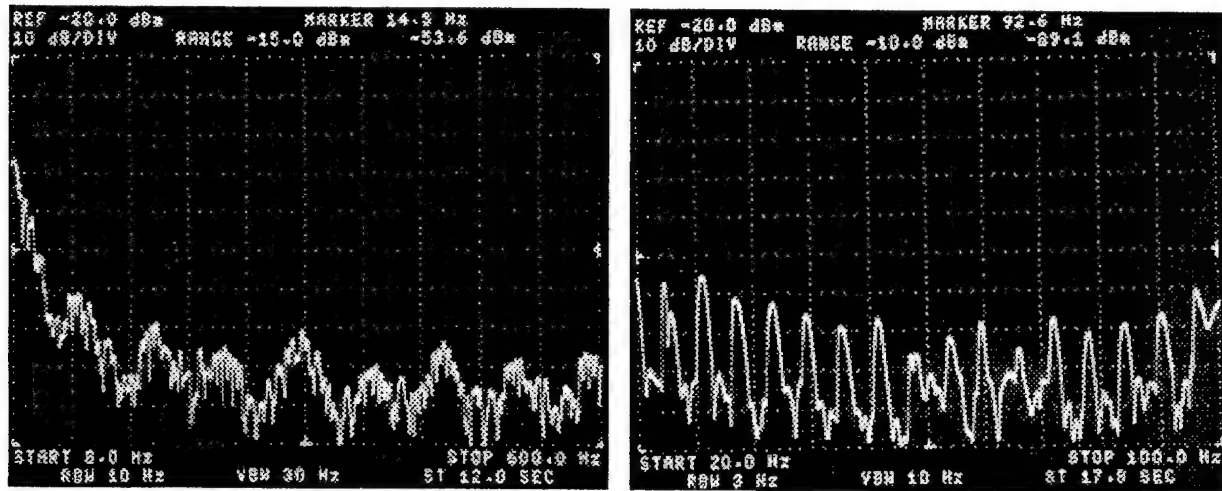
Figure 3.5: Power Spectrum of the Filtered Accelerometer Sensor Output
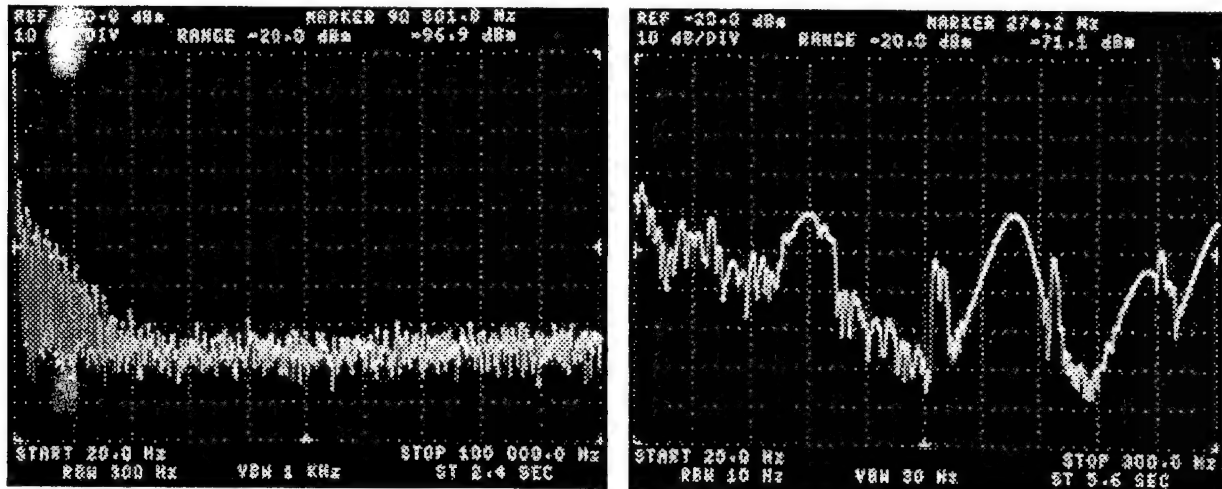
Figure 3.6: Power Spectrum of the Filtered Angular-rate Sensor Output

This short-fall is more apparent in the angular-rate sensor outputs than those of the filtered accelerometer outputs. The low-pass filter appears to be doing an acceptable job in attenuating the noise energy in the accelerometers. However, the low-pass filter is not fully attenuating the noise energy in the angular-rate sensors to a level below the resolution of the least significant bit (LSB). Specifically, there are noise peaks at 30, 100, and 275 Hz that reach -60dB. Clearly, these high energy noise peaks are affecting the LSB. By noticing that these noise peaks represent a 12dB over-power above the required noise floor, and solving the following equation for N:

$$20\log\frac{1}{2^N} = -12dB$$

$$N = 1.99$$

it's apparent that this noise energy is actually affecting the two least significant bits of the measured angular-rate signal.

During system development, it was discovered that the x-angular-rate signal being input to the A/D had apparent slow growth behavior. Figure 3.7 shows the results of sampling the x-angular-rate sensor once every 60 seconds for a period of 4 hours after starting the SANS from an initially "cold" state.
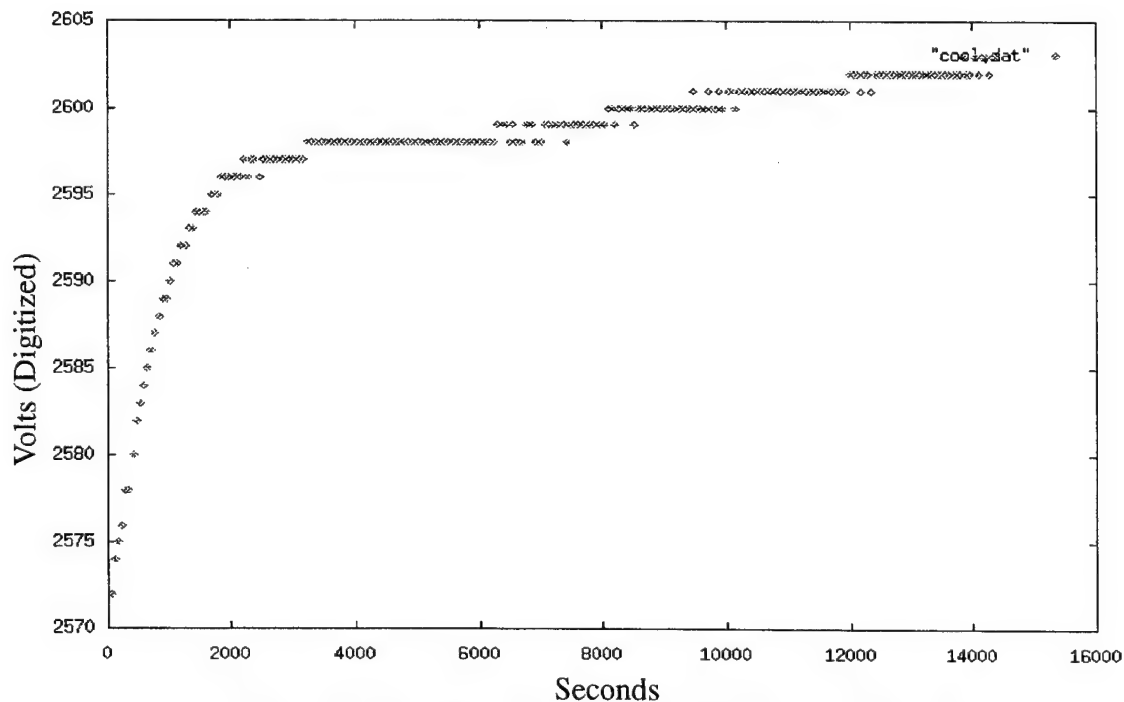
Figure 3.7: Sampled X-Angular-Rate Over a 4-Hour Period

The system hardware configuration used to collect the data shown in Figure 3.7 included the the LPF circuitry described in [Bachmann 95]. This growth in the signal turned out to be a significant discovery possibly explaining some of the inaccuracy the SANS had been experiencing during previous testing. It was at first decided that this growth was due to heating effects in the IMU. However, after replacing the LPF circuitry with a newly designed circuit using commercially provided filters (this new filter circuit is described later in Chapter IV), the problem went away. Therefore, the growth error in the x-angular-rate signal was attributed to heating effects on the circuit components conditioning the x-angular-rate signal in the LPF circuitry used in the prototype SANS.

D. SUMMARY

This analysis lends further explanation to the results obtained in [McGhee 95], which shows the angular-rate output fluctuating as much as 4.88 mV; the three least significant bits of the 12-bit measured signal. Some of this fluctuation could be attributed to any of the causes stated in [McGhee 95], but this analysis gives qualitative evidence that the two least significant bits of the measured angular-rate data were in error due to sensor noise.

IV. SYSTEM HARDWARE CONFIGURATION

A. INTRODUCTION

Figure 4.1 presents a block diagram for the hardware making up the redesigned SANS. Figure 4.2 presents a photograph of the SANS components fully assembled into their testing configuration. The project box in which the components are currently mounted is not intended to be the final resting place for these components. A more permanent, water-tight, streamlined housing is currently in development. In fact, by using this particular off-the-shelf project box, and by iteratively installing, testing, changing, and then reinstalling the components, an optimum space-efficient configuration has been achieved, which, in turn, has driven the design of the final housing.

This configuration is significantly different from that presented in [Bachmann 95] for the previous prototype. Mainly, the SANS components are no longer separated; all its components are physically located in one, self-contained package. In fact, when joined with its accompanying power source (a 12 VDC battery), it is capable of being strapped-down to a testing turntable, or inserted into the towfish (with slight towfish modifications) described in [Bachmann 95]. In its current configuration, the SANS has its processor and GPS/DGPS components "onboard," thus no longer requiring the transfer of sensor data via modem to an external processor or GPS/DGPS receiver as was shown in [Bachmann 95]. In order to maintain the ability for human monitoring and interaction during the course of an experiment, the SANS's processor is linked with an external processor via a DOS TCP/IP network environment. This external processor's only function is to maintain a remote control session with the SANS processor and receive its attitude and position updates. Contrary to the original SANS proof of concept design presented in [Bachmann 95], the SANS now maintains the capability to on-board-process its own data in order to maintain its general capability to interface with any other higher-level processor (via a network), regardless of the application.

B. HARDWARE DESCRIPTION

1. Computer

The on-board processor is an Extremely Small Package (E.S.P.) Cyrix 486SLC DX2 50 MHz computer. This computer, pictured in Figure 4.3, is specifically designed to offer off-the-shelf PC-compatible solutions in space and/or power constrained environments. Together, the E.S.P.
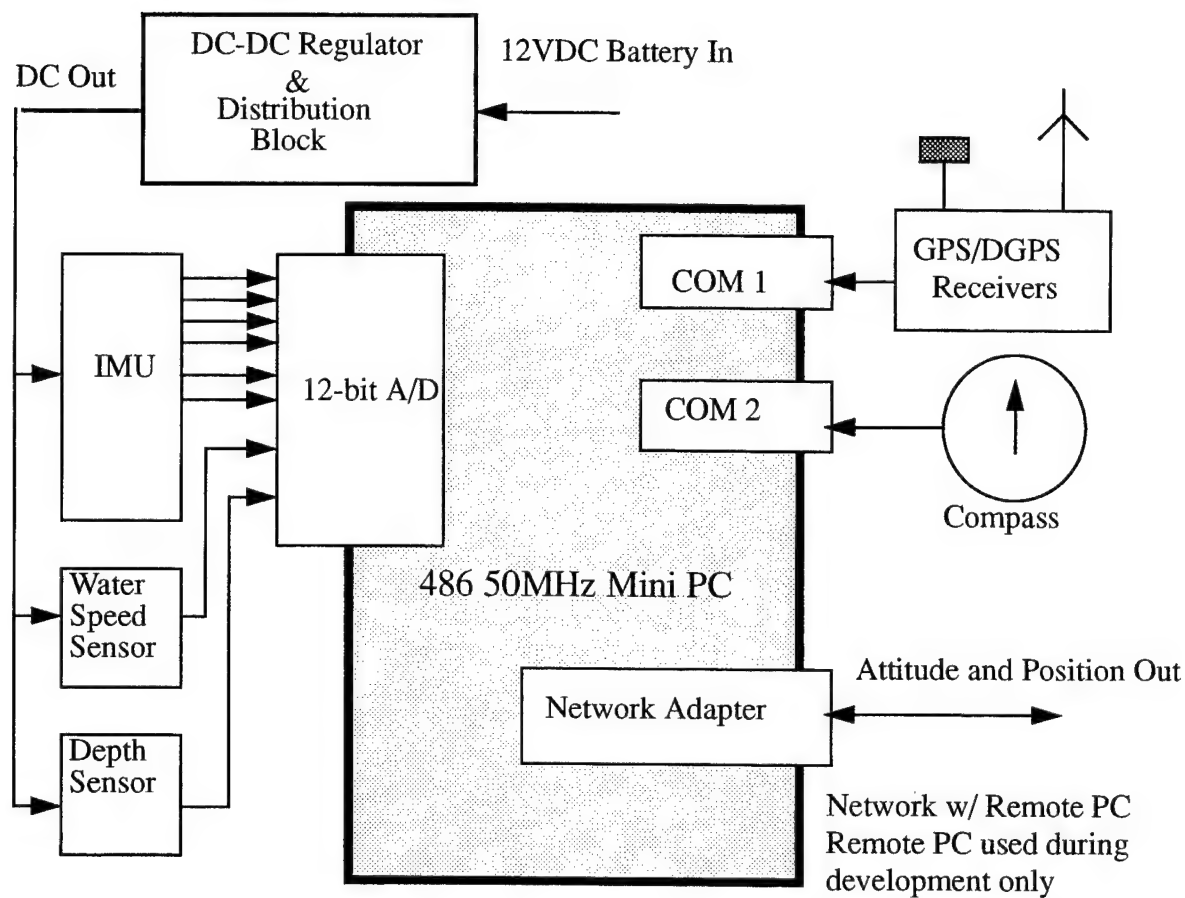
Figure 4.1: Block Diagram of the Redesigned SANS Hardware Configuration

Processor Module and its accompanying modules provide a small, low-power system that does not sacrifice system performance compared to a standard, desk-top type system [MAXUS 95]. This particular E.S.P computer possesses a total of eight modules which perform various system tasks. A brief description of the tasks these various modules perform in the SANS is as follows:

a. 486SLC DX2 CPU Module

Besides providing the processing capability, the CPU Module provides the interface for a standard keyboard, the Flash PROM containing the system BIOS, and memory and bus controller logic [MAXUS 95].
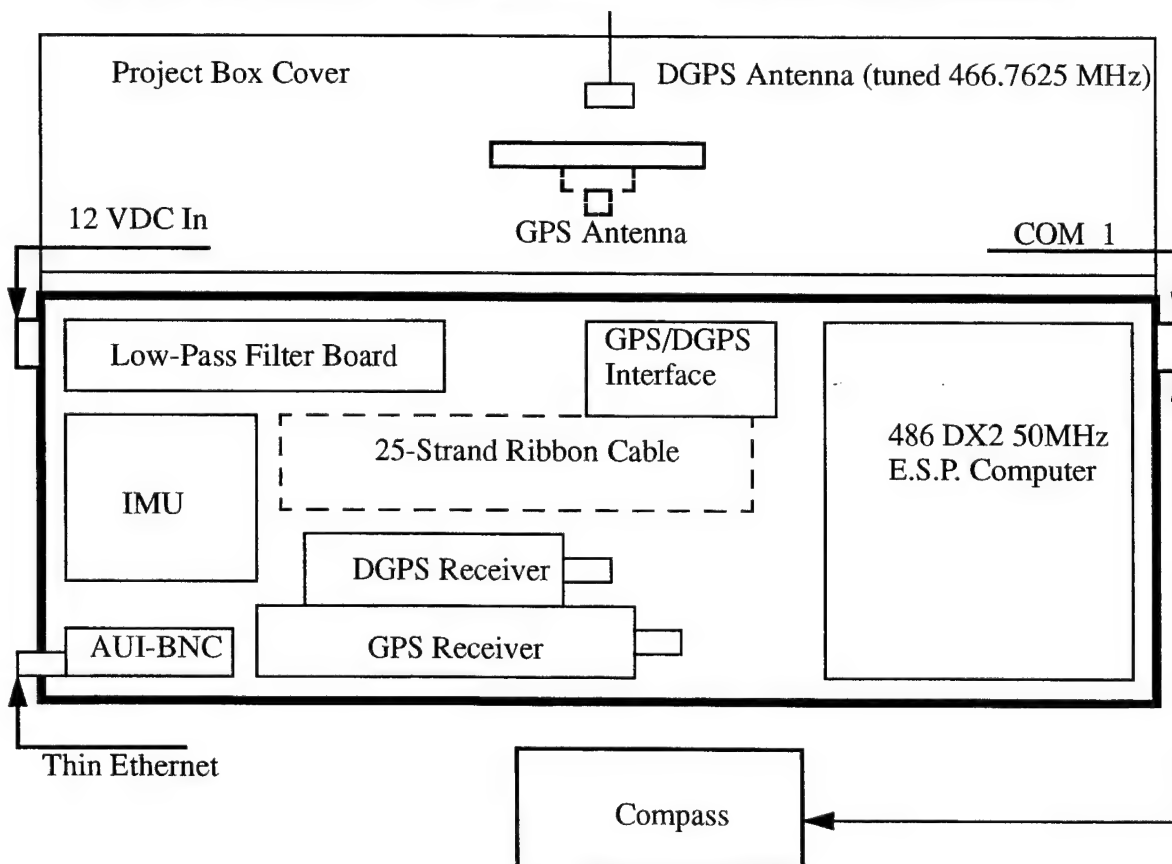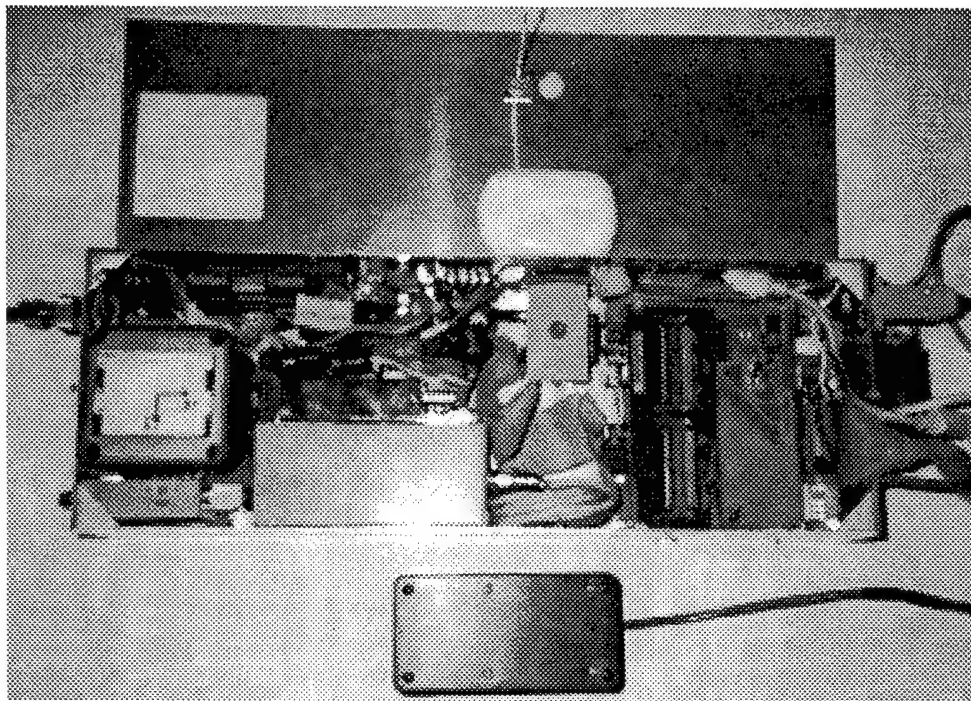
Figure 4.2: SANS Hardware Configuration
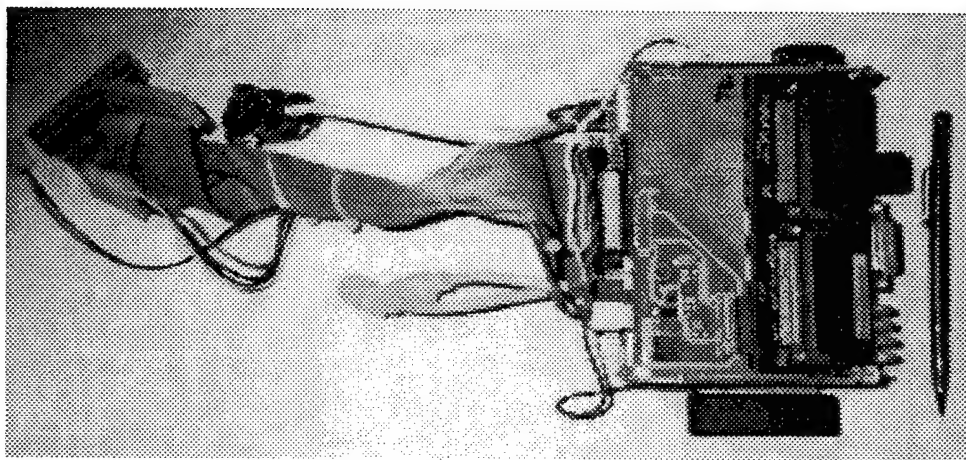
Figure 4.3: E.S.P. 486SLC DX2 50 MHz Computer

b. DC-DC Power Module

Provides for all the system power requirements up to a maximum 35W total output. It accepts an unregulated 12 VDC and provides the required +5, +12, -12, and -28 VDC to power various system components and optional peripherals (i.e., external floppy/hard drive). [MAXUS 95]

c. VGA Adapter Module

Provides the interface to operate an external VGA monitor.

d. PC I/O Module

Provides for 2-Serial ports and 1-Parallel I/O port.

e. PCMCIA Module

Provides two type-III PCMCIA sockets which conform to PCMCIA Release 2.01 standard [MAXUS 95]. These two ports can be used for a variety of compatible devices (i.e., Ethernet Adapter, Modem, GPS Receiver, etc.). For instance, this module can accept SRAM cards which can contain bootable system files or simply be used to provide storage media. This module was included in the current design to provide additional secondary storage in the form of PCMCIA SRAM cards, as well as to enable future expandability.

f. Ethernet Module

Provides the SANS with an external ethernet interface.

g. Analog to Digital (A/D) Module

Provides 8 differential or 16 single-ended input channels at 12-bit resolution. It features a single-channel maximum sampling rate of 333 KHz, and an input range from +/- 1.25mV to +/- 10V [MAXUS 95]. The A/D module provides a 34-pin external connector (J3) to which developers can connect their input signals. In its current configuration, the A/D module samples only 8 of the available 16 single-ended channels. Table 4.1 shows the current pinout of connector J3 on the A/D connector board.

Signal	Pin	Pin	Signal
GND	1	2	GND
GND	3	4	IN_8B unused
GND	5	6	IN_8A depth
GND	7	8	IN_7B unused
GND	9	10	IN_7A water speed
GND	11	12	IN_6B unused
GND	13	14	IN_6A z-axis angular-rate
GND	15	16	IN_5B unused
GND	17	18	IN_5A y-axis angular-rate
GND	19	20	IN_4B unused
GND	21	22	IN_4A x-axis angular-rate
GND	23	24	IN_3B unused
GND	25	26	IN_3A z-axis acceleration
GND	27	28	IN_2B unused
GND	29	30	IN_2A y-axis acceleration
GND	31	32	IN_1B unused
GND	33	34	IN_1A x-axis acceleration

Table 4.1: Connector J3 Pinout

h. DRAM Module

Provides for high-speed (70ns) memory storage available in 2, 4, 6, 8, or 16MB capacities [MAXUS 95]. This module is to the E.S.P. as a hard disk is to a standard desk-top PC.

2. Inertial Measuring Unit

The inertial navigation component of the SANS is provided by a Systron-Donner Model MP-GCCCQAAB-100 "MotionPak" inertial sensing unit, pictured in Figure 4.4. This self-contained unit provides analog measurements in three orthogonal axes of both acceleration and angular velocity. It consists of a cluster of three accelerometers and three "Gyrochip" angular rate sensors. General specifications are shown in Table 4.1. Accelerometer specifications and angular rate specifications are shown in Table 3.1 and Table 3.2 respectively.
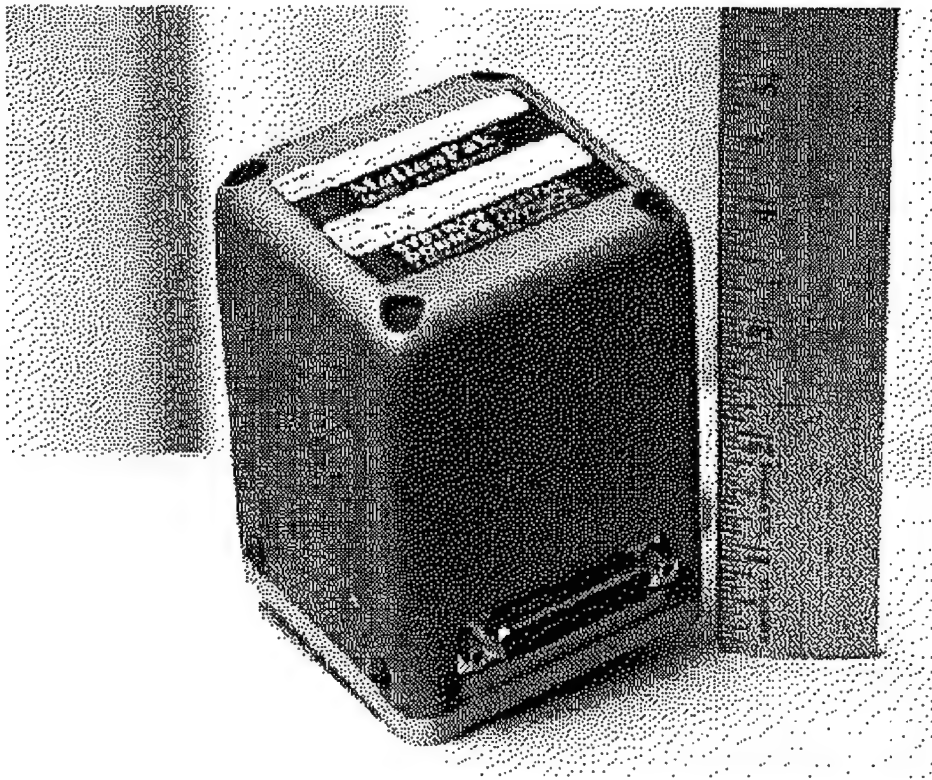
Figure 4.4: Systron-Donner Inertial Measuring Unit [Bachmann 95]

Parameter	Units.	Range
Input Voltage	DC Volts	+15, -15
Input Current	Amps	+0.246, -0.196
Temp. Range	degrees C	-40, 80
Weight	grams	912
Temp. Sensor	μ A/deg k	1.0

Table 4.2: MotionPak General Specifications [Sytron-Donner 94]

3. GPS/DGPS Receiver Pair

The GPS/DGPS receiver used is the ONCORE 8-channel receiver which incorporates an imbedded DGPS capability [Oncore 95]. The receiver is capable of tracking up to eight satellites simultaneously. It can provide position accuracy of better than 25 meters Spherical Error Probable (SEP) without Selective Availability (SA) and 100 meters (SEP) with SA. Typical Time-To-First-Fix (TTFF) is 18 seconds with a typical reacquisition time of 2.5 seconds [Oncore 95]. This receiver meets or exceeds the capabilities of the receiver described in [Norton 94], which demonstrated that under normal operating conditions, a receiver of this kind, is capable of meeting the accuracy and time requirements of the SANS project. [Norton 94] also demonstrated that a receiver with these qualities will perform well when using an antenna that is located on or near the sea surface as is necessary during a clandestine mission. Figure 4.5 shows the ONCORE GPS/DGPS receiver used in the SANS project.
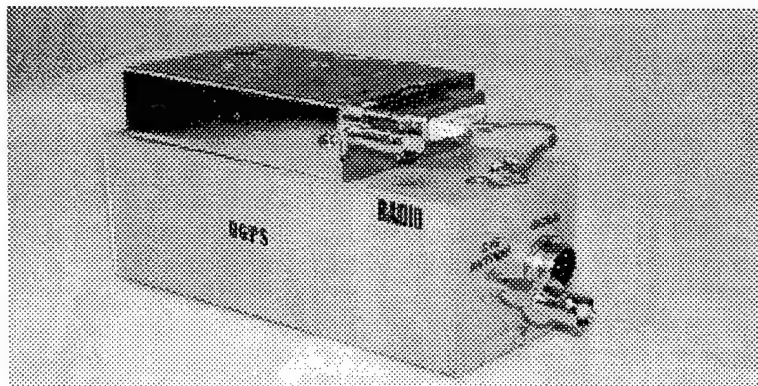
Figure 4.5: ONCORE GPS/DGPS Receiver

31

4. Low-Pass Filters

Based on the analysis given in Chapter III, the 2-pole Bessel filters were replaced with new filters which offer a faster "rolloff" in the stopband (now 80 dB/Dec). Each of the six IMU outputs is filtered by a 4-Pole, active, anti-aliasing low-pass Butterworth filter with a bandwidth of 10 Hz. The low-pass filters are model DP74 and are packaged in a standard 16-pin dual in-line package (DIP) [Frequency Devices 96]. These filters feature low-harmonic distortion, come factory tuned to a user-specified corner frequency, require no external components or adjustments, and operate with a dynamic input voltage range from non-critical +/- 5V to +/-18V power supplies [Frequency Devices 96]. To implement these filters into the SANS, a double-sided printed circuit board (PCB), shown in Figure 4.6, was designed and machined to receive all six filter DIPs as well as three quad op-amp LM324 DIPs configured as voltage-followers to provide input and output circuit protection. Figure 4.7 shows the schematic diagram for one channel in this low-pass filter circuitry.
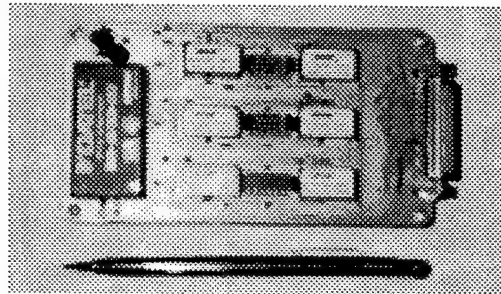
Figure 4.6: The Double-Sided Low-Pass Filter PCB
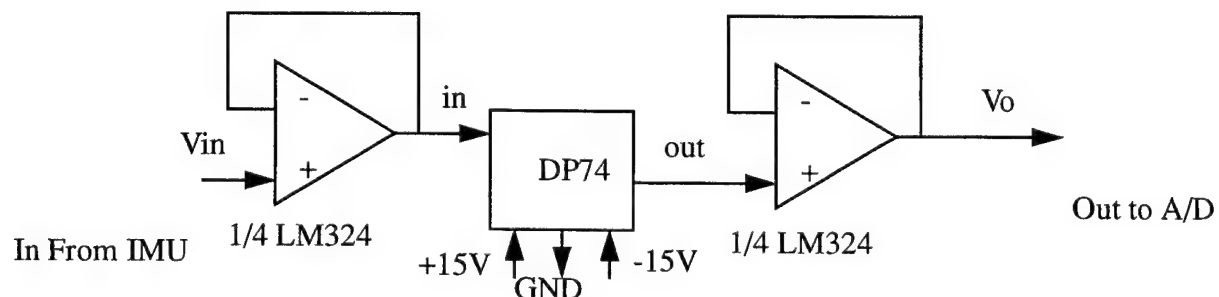
Figure 4.7: Schematic Diagram for One Channel of the Low-pass Filter Circuitry

5. DC-DC Converter

To provide for the requisite +/-15 VDC, a DATEL model BWR-15/330-D12 DC-DC Converter is used to convert the unregulated 12 VDC battery input to regulated +/-15 VDC to power the low-pass filter circuits as well as the IMU. This converter features over-current and short-circuit protection, a compact form-factor, high reliability, a minimum efficiency of 82%, and employs switching regulator technology, which minimizes heat generation and current usage [DATEL 95]. Though the DC-DC converter ensures a low noise/ripple in the output signal, the converter is augmented with additional capacitors in parallel with both the input and output pins of the device. Figure 4.8 shows the use of these capacitors in the DC-DC converter circuitry.

C1: 0.1 uF Electrolytic Capacitor
C2: 100 uF Electrolytic Capacitor
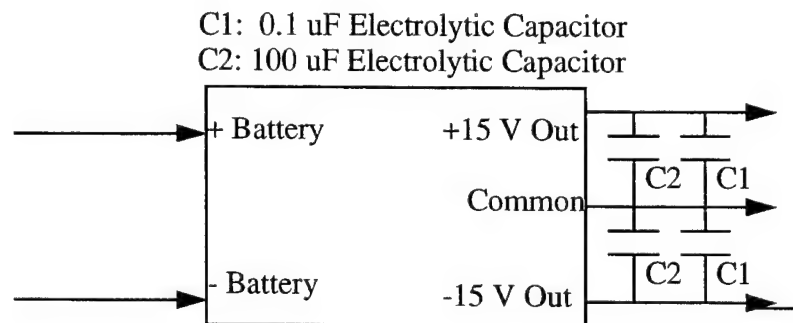
Figure 4.8: DC-DC Converter Circuit

The value of C1 was selected so as to filter any high-frequency radio frequencies (RF) that may get inducted into the circuit from surrounding components, and the value of C2 was selected so as to filter any high-frequency switching noise from the converter itself (the converter switches at 165 KHz) [DATEL 95]. This DC-DC Converter is physically mounted on the same PCB on which the low-pass filters are mounted. This configuration allows the use of PCB "tracks" to supply the filter circuitry with +/-15VDC. This regulated +/-15VDC as well as IMU sensor signals are routed to and from the PCB via a DB25.

6. Ribbon Cable

Physically connecting the IMU, Low-pass Filters/DC-DC Converter PCB, the Analog-Digital Converter, input power, water speed sensor, and depth sensor, is a 25-strand flat ribbon cable. This cable enables all system components to be easily interconnected. Table 4.2 gives the pinout of this 25-strand ribbon cable as well as all DB25 connectors used in the system..

Pin Number	Color	Description
1	Black	IMU x-axis acceleration output
2	Gray	x-axis acceleration input to A/D
3	Blue	IMU y-axis acceleration output
4	Yellow	IMU z-axis acceleration output
5	Red	z-axis acceleration input to A/D
6	Black	unused
7	Gray	unused
8	Blue	-15VDC
9	Yellow	GND (Ground)
10	Red	unused
11	Black	+15VDC
12	Gray	unused
13	Blue	Depth sensor input to A/D
14	White	y-axis acceleration input to A/D
15	Purple	IMU z-axis angular-rate output
16	Green	z-axis angular-rate input to A/D
17	Orange	+12 VDC Battery In
18	Brown	Water speed sensor input to A/D
19	White	unused
20	Purple	y-axis angular-rate input to A/D
21	Green	x-axis angular-rate input to A/D
22	Orange	IMU x-axis angular-rate output
23	Brown	IMU y-axis angular-rate output
24	White	unused
25	Purple	unused

Table 4.3: Pinout of the Ribbon Cable and DB25 Connectors

7. Compass

The compass used in the SANS project is a Precision Navigation model TCM2 Electronic Compass Module. This compass does not employ the mechanical gimbal technology utilized in the compass described in [Bachmann 95], but rather employs a three-axis magnetometer and a high-performance two-axis tilt sensor in a small form-factor [TCM2 95]. The TCM2 compass provides readings of not only heading, but also pitch, role, and surrounding magnetic field strength. The TCM2 provides greater accuracy by calibrating (performed by the user) for distortion fields in all tilt orientations, providing an alarm when local magnetic anomalies are present, and giving out-of-range warnings when the unit is being tilted too far [TCM2 95].

8. Other Components

The water speed sensor and the depth sensor are those described in [Bachmann 95] and therefore not depicted in Figure 4.2. The GPS antenna shown in Figure 4.2 is an active antenna, which was selected for its performance and low-profile. Because the E.S.P. Ethernet module's output media type is AUI, a standard AUI-to-BNC media converter is employed to allow the use of durable RG-58 coax cable to span the roughly 100m distance required while pulling the towfish behind a towing vessel. The GPS/DGPS Interface box is nothing more than an adapter to interface the GPS receiver signal with the serial COM2 port of the E.S.P. computer.

C. SUMMARY

The SANS design described in this chapter is significantly different from that described in [Bachmann 95]. The processing capability, along with the GPS/DGPS receiver, is now on-board the SANS, making it completely self-contained with its only external link being that of a DOS ethernet environment to a remote PC. The IMU sensor data, after low-pass filtering, along with water speed and depth data, are converted from analog to digital with 12-bit resolution, and then joined with GPS data in the on-board computer which computes updated attitude and position information to be exported over an ethernet socket. The hardware for this version of the SANS was chosen to comply with the requirements set forth in [Kwak 93]. Though there are many possible choices of hardware for each of the components in Figure 4.2, trade-offs between accuracy, size, power requirements, and cost must be considered. As further advances in miniaturization are

made, accuracy will continue to increase while price and size decrease, thus making it easier to meet the challenges of the SANS baseline requirements.

V. SOFTWARE DEVELOPMENT

A. INTRODUCTION

The purpose of the SANS software remains the same as that described in [Bachmann 95]: "to utilize IMU, heading, and water-speed information to implement an INS, and then integrate this with GPS information into a single-system which can produce continually accurate navigational information in real time." This chapter will not re-introduce the software already adequately presented in [Bachmann 95], but rather will present only those changes, additions and updates, to the software described for use on the previous prototype SANS.

B. SOFTWARE DESCRIPTION

This implementation continues to use a majority of the software designed by [Bachmann 95], and for the most part, unchanged. However, the changes in the hardware architecture have driven subsequent changes to the software design to enable its use in the current SANS. Figure 5.1 shows the software objects and data flow of the current SANS. Though already previously stated, these hardware changes, along with the subsequent software changes and additions, are again presented in detail as follows:

1. Compass Data

In the SANS described by [Bachmann 95], the compass data was included in the packets received via modem from the towfish. This compass data was then parsed out of this packet for use. This Xmodem packet code is no longer required and has been removed. In the current version of the SANS, the compass data is received via the COM2 serial port, and thus requires code to communicate with the serial port, as well as code to check the "checksum" and header of each compass message received. This change was easily implemented by simply cutting, pasting, and altering previous com port and checksum code. This code is provided in Appendix A.

2. GPS Data

The code required to process GPS information is only slightly different from that described by [Bachmann 95]. The only changes required were driven by the use of an 8-channel receiver vice the 6-channel GPS receiver used previously. The 8-channel receiver sends a longer message, thus the only changes were to adjust the message length and the location of the checksum character in

the GPS message.

The code was also modified to include a differential check. Before the code recognizes a GPS message as being valid, the message must pass three conditions; 1) A checksum check, 2) The fix must be based on at least 4 satellites, and 3) The differential bit in the message must be set (i.e., the fix must have the differential correction calculated into it). This updated code is provided in Appendix A as well.
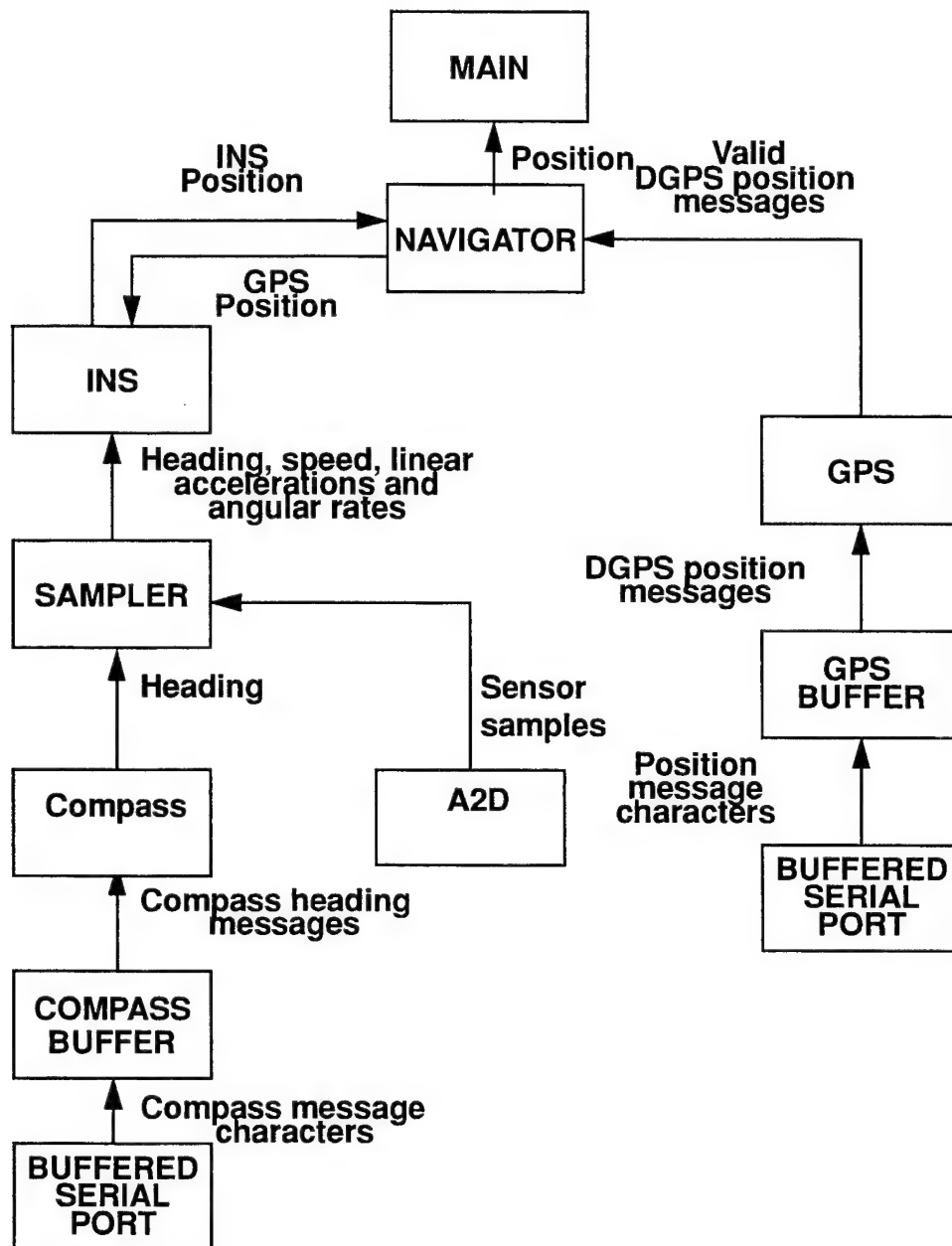
Figure 5.1: SANS Software Objects and Data Flow

3. Inertial Sensor Data

Like the compass data, the inertial sensor data was previously included in the packets received via modem from the towfish. Currently however, the filtered inertial data is input directly to the A/D converter module in the on-board processor. Therefore, there is additional code running in the SANS software system which operates the A/D converter module, as well as buffers this data.

a. A2D

The A/D module, when delivered, came with C source code to run a demo of the module. This code was modified and converted to C++ to fit the SANS application and its object oriented design. The A2D class provides all the requisite software operation of the A/D module in the E.S.P. computer. The A/D module is completely controlled through software. Primary to this control is the manipulation of the A2D Control Register and the A2D Status Register. These registers are manipulated by writing to and reading from specific memory addresses. Specifically, the A2D Control Register is at address 0x108, and the A2D Status register is at 0x102. The A2D class is designed such that the user maintains some degree of flexibility. For instance, the user can choose between one of two base addresses, 0x100 or 0x300, from which these important A2D registers are then created by adding an offset (08h and 02h respectively) to this base address.

Though the A2D class has many member functions, the SANS software only uses a few in accomplishing its mission. All those member functions not directly utilized in this particular application are useful to troubleshoot problems with the A2D module, or allow a wide range of options to tailor its use to a particular application. The class code is well commented, and easily stands without any further discussion. However, for the benefit of the reader, the following general discussion of how the A2D module works in the SANS application is deemed beneficial.

The A2D, as stated in Chapter IV, provides 12 bits of resolution, or $2^{12} = 4096$ discrete quantization levels. There are two modes to employ the A2D module: differential mode, or single-ended mode. The SANS application employs the A2D in the single-ended mode of operation. In this mode, the A2D is able to sample the dual-ended swing of the IMU sensor signals, and represent these voltages as a digital value in the range 0 - 4095. A general A2D conversion table is provided as Table 5.1 to further illustrate how the sensor voltages are mapped over to their digital equivalents.

Sensor DC Voltage	Converted Equivalent
+10 Volts	4095
+5 Volts	3071
0 Volts	2047
-5 Volts	1023
-10 Volts	0

Table 5.1: A2D DC-to-Digital Conversion Mapping

When an A2D object is instantiated, the class constructor (see Appendix A, A2D.cpp) sets several default data member values, and then reads the A2D configuration file A2D.cfg. This configuration file provides a simple manner in which a user can change the way the A2D module operates without having to re-compile the source code. When the function readConfigFile() is called, it reads the A2D.cfg file one line at a time and loads those respective variables described on each line of this file with those values found on each line. This prepares the software for initializing the A2D hardware. The constructor initializes the system addresses to setup for A2D operation, then initializes the A2D hardware using those variables that were loaded upon reading the configuration file. The A2D object gets instantiated when a Sampler object gets instantiated as the A2D object "a2d1" is a private data member of the Sampler class.

The A2D module gets set into operation by a call to initSampler(). This sampler class member function executes a sequence of function calls to A2D class member functions which set the A2D module into operation. Mainly, they program the sequencer to tell it which channels to sample in which order, reset the A2D First-In-First-Out (FIFO) to enable it to receive data, and then toggle the trigger bit in the A2D Control Register from a low to a high which starts the A2D into operation.

During SANS operation, the Sampler class member function readSamples() is called repeatedly to retrieve inertial data from the A2D FIFO. It first checks to ensure that the FIFO is not FULL. If the FIFO ever gets filled without being immediately emptied, it will continue to push data into the FIFO. Since there is no room for additional data, all samples from that point on will be lost. So, it is evident that preventing the FIFO from overflowing is paramount to SANS operation. If the this check is ever true, the SANS software will terminate execution. To prevent

FIFO overflow, one need only be mindful of the rate at which the A2D is sampling its inputs and be sure to empty out the FIFO at the same rate or faster. If the FIFO does have data in it, this data is emptied from the FIFO and stored in a doubly-subscripted array with 8 rows and 1000 columns to coincide with storing up to 1000, 8 channel samples of sensor data. Figure 5.2 presents a model of this doubly-subscripted array and how it stores the data.

x-acc	x-acc	. . .
y-acc	y-acc	. . .
z-acc	z-acc	. . .
x-ang	x-ang	. . .
y-ang	y-ang	. . .
z-ang	z-ang	. . .
waterspeed	waterspeed	. . .
depth	depth	. . .

0   1 999

Sample Number/Array Index

Figure 5.2: Model of the A2D Sample Array

As the samples are being emptied from the FIFO, the variable "timeCounter" is incremented once for every 8 samples that are pulled from the FIFO. This variable is then multiplied by the sample period to calculate the "deltaT" or the time between adjacent samples. The reasoning behind using this type of data structure to temporarily store the data is to enable access to a history of samples in order to employ a digital filtering scheme. In the case of the SANS, it employs a very simple form of low-pass filtering by averaging over all the samples received since the last sample was taken from this array.

C. SUMMARY

All software additions and updates to the SANS software were made in keeping with object-oriented paradigms. The software was written in Borland 3.1, C++ for DOS.

Since the publication of [Bachmann 95], there have been many significant as well as minor changes/updates to the SANS software. The compass data is now received from a serial port vice being received via an Xmodem packet. The GPS data is still received from a serial port, but the GPS receiver is now an 8-channel receiver instead of a 6-channel. The inertial, water speed, and

41

depth data is no longer being received via an Xmodem packet, but rather is being input directly into the A/D module in the E.S.P. computer. Consequently, all the Xmodem packet code is no longer used in the SANS software system. All these hardware changes have driven software changes that have propagated throughout the software. For this reason, a complete copy of all SANS software is given as Appendix A. A further discussion of those software updates incorporated into the SANS between the publication of [Bachmann 95] and this thesis, is presented in [Bachmann 96].

VI. SYSTEM TESTING

A. INTRODUCTION

This chapter presents the test method and the experimental results of testing used to determine the functionality and accuracy of the SANS. Bench testing was performed to ensure the entire system was functioning properly in its current state. The system was then tested by conducting tilt-table tests in order to verify the operation of the Inertial Measurement Unit and its associated filter software. It was further tested by conducting a static GPS test to test the operation of the positioning capabilities of the SANS. Lastly, it was given a static heading test to ensure proper operation of the Kalman filter in determining heading.

Figure 6.1 presents a data flow diagram of the SANS navigation filter. This diagram is presented at this point to provide a basis for discussion of the following test results. The reader is referred to [Bachmann 95, 96], and [McGhee 95] for an in-depth discussion of this filter. This twelve-state velocity-aided navigation filter is implemented in the SANS software provided as Appendix A and Appendix B.

B. IMU TESTS

The purpose of these tests was to qualitatively ensure that the SANS was able to accurately track changes in attitude. Paramount to getting the SANS to do this, is to find a suitable value for the gain K1(shown in Figure 6.1), the biasWght, sampleWght (all in INS.CPP), and x/yAccelScale factors (in LOCATION.H). The results presented in this chapter are for the pitch axis only. A similar process still remains to be done for the roll axis.

With the SANS mounted to the tilt-table described in [Bachmann 96], a series of pitch tests were conducted. For these tests, the IMU outputs were sampled at a rate of 40 Hz. During testing, it was noticed that the SANS produced update rates between 6-10 Hz.

Tuning data for the SANS was obtained by moving the SANS unit through pitch excursions within a 45 degree range. The attitude as determined by the SANS was then plotted and compared with the actual motion of the unit. Through this comparison, it was possible to determine an initial scale factor. This process was repeated several times until the attitude determined by the filter 'matched' the true motion of the unit. From this analysis, a rate sensor scale factor of 4.1 enabled the filter to register a pitch of 45 degrees when the SANS was pitched to that angle.
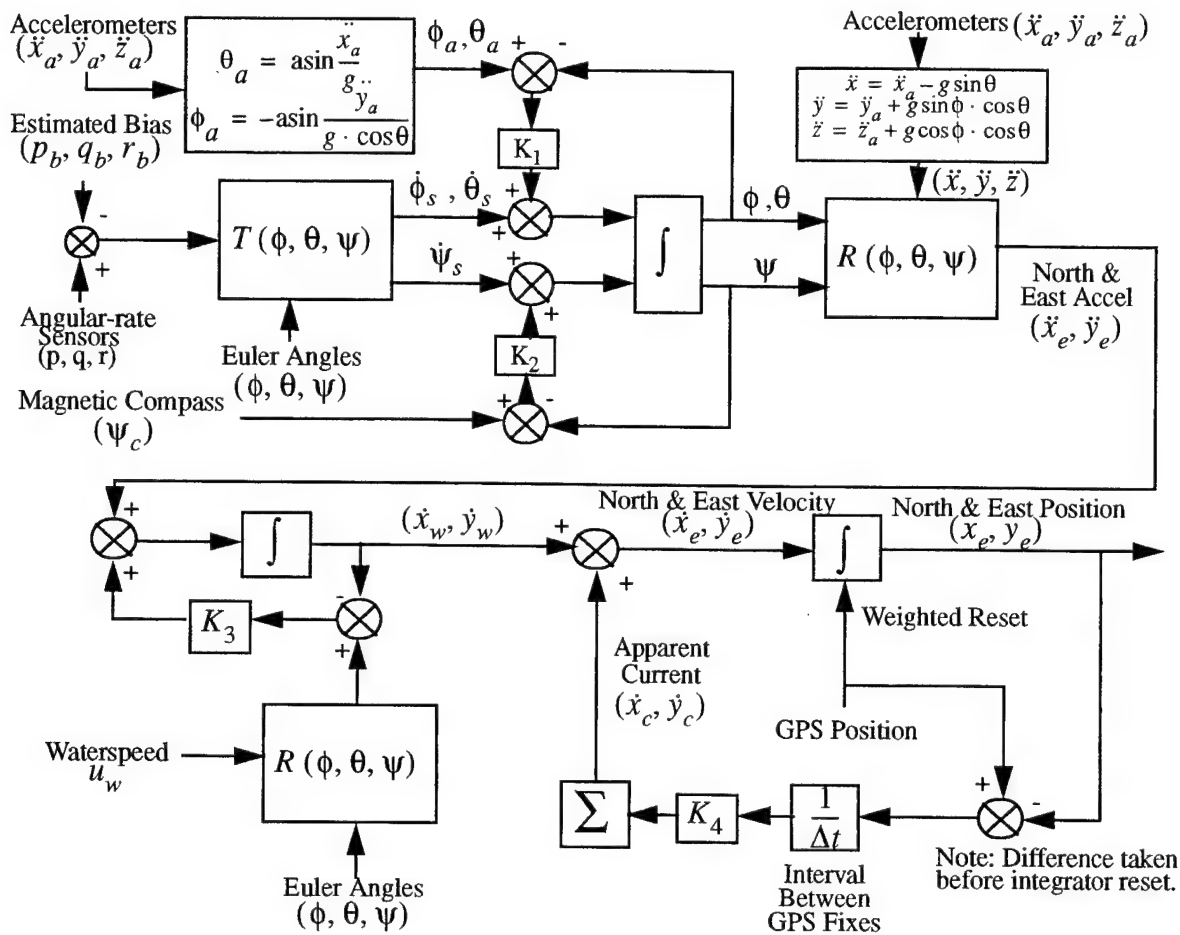
Figure 6.1: SANS Navigation Filter

In order to tune the filter for accurate operation, the rate sensor bias value must first be determined. To do this, the gain value K1 is set to zero in order to prevent accelerometer data from getting to the first integrator. Without the accelerometer data, only the high frequency data from the angular-rate sensors is input to the first integrator with the estimated bias. Taking the commanded tilt-table angles to be truth then, any errors in attitude can be attributed to the bias and scale factor.

Having determined the initial scale factor to be 4.1 and setting K1=0.0, a series of pitch tests were conducted in order to determine the correct bias weights. In order to obtain a starting point for this analysis, and based on similar testing discussed in [Bachmann 95], an initial bias value (biasWght) of 0.999 was chosen for the first pitch test. Figure 6.2 shows the results of the first 45 degree pitch excursion with K1 = 0.0, bias value = 0.999, and the scale factor set to 4.1.
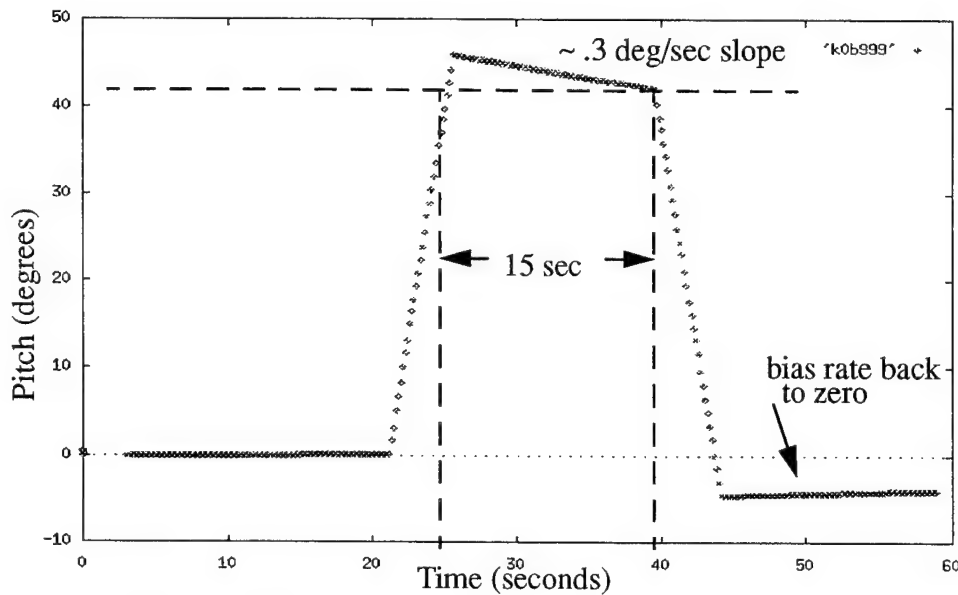
Figure 6.2: 45 Degree Pitch Excursion w/K1=0, biasWght=.999, scale factor= 4.1

The shape of the plot in Figure 6.2 is determined by the value of a "virtual" time constant. For the sake of this discussion, this time constant is τ. The method in which the SANS software determines the rate bias value can be essentially modeled as a low-pass filter. Specifically, the SANS software figures this rate bias according to the model depicted in Figure 6.3. This figure describes a linear system where the accumulating rate bias is subtracted from each new angular-rate value, multiplied by the sampleWght ($(\Delta t)/\tau$), which is (1 - biasWght), and then this value is added to the rate bias sum.
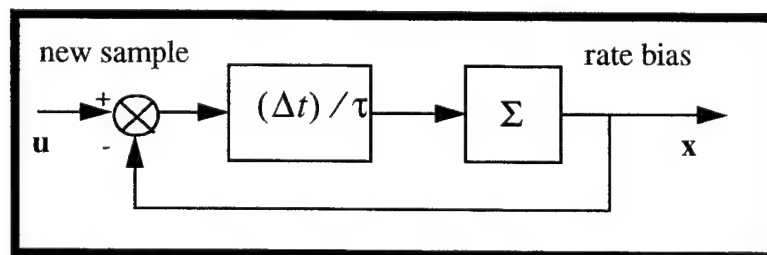
Figure 6.3: Model of Rate Bias Low-pass Filter

45

From linear system theory, in the s-domain, it can be shown that the transfer function of this system is:

$$\frac{1}{1 + \tau s}$$

Assuming the 10 degree/second pitch excursion to be a unit step function, the unit step response of this system is:

$$x(t) = \left(1 - e^{-t/\tau}\right)$$

By taking the first derivative, it can be shown that the bias filter output slope is $1/\tau$. Using the relation that the sampleWght (.001 for the first test) is equal to the ratio of the time between samples (at a sample rate of 7 Hz, this gives a period of 0.142) and τ, it can be shown that, for the first test depicted in Figure 6.2, τ is approximately 142 seconds. That is,

$$0.001 = \frac{\Delta t}{\tau} = \frac{1/7}{\tau} \Rightarrow \tau = 142 \text{ sec}$$

Given the known angular-rate of the pitch excursion was 10 degrees/second, the bias filter output slope can then be calculated as:

$$\frac{10}{142} = 0.07 \text{ (deg/sec)/sec}$$

From the plot, it appears to have taken about 4.5 seconds to complete the 45 degree angular-rate excursion. Multiplying the filter slope by the 4.5 seconds yields a resulting bias estimate of approximately 0.3 deg/sec. Again, from the plot, the slope of the curve after it reached 45 deg, but before it started its return to zero, agrees with the calculations.

In short, the filter is behaving just as it should. During the course of the excursion to 45 degrees, the bias filter accumulates a rate bias of 0.3 deg/sec. Since the filter time constant is only 142 seconds, and the SANS only underwent a 4.5 second excursion, it did not have time to correct for this bias. Additionally, since the bias is subtracted from each new sample, the result is the negative sloping portion of the curve while the SANS is at 45 degrees pitch.

Obviously, the goal of this analysis is to minimize any accumulated rate bias while the SANS is undergoing pitch or roll excursions. To accomplish this, the value of the time constant τ needs to be long enough to minimize the bias filter slope, which in turn, minimizes the accumulated rate bias.

46
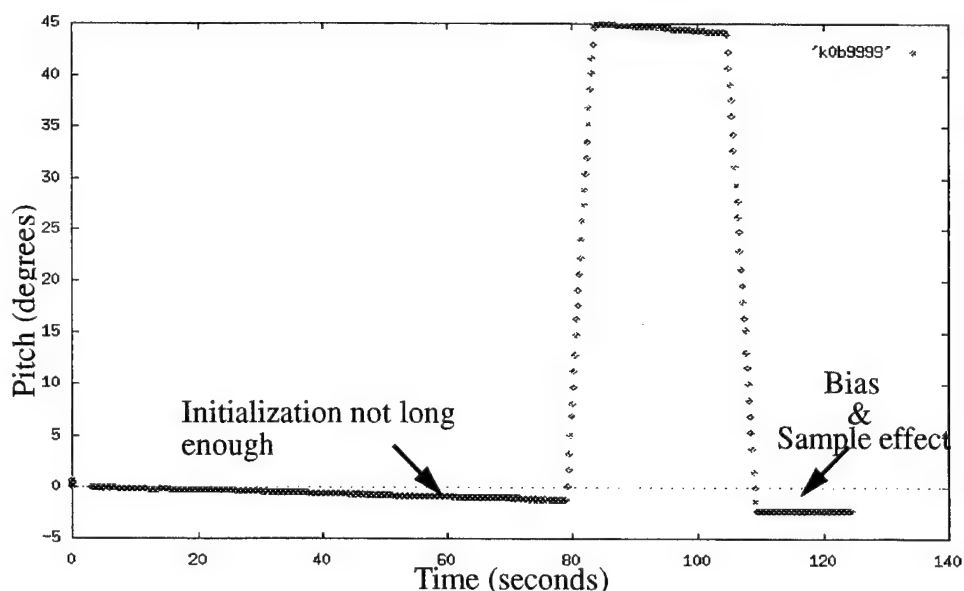
Figure 6.4: 45 Degree Pitch Excursion w/K1=0, biasWght=.9999, scale factor= 4.1

Figure 6.4 shows the test results after increasing the filter time constant by a factor of 10. As shown in the plot, the pitch values sloped off less by about a factor of 10 (as expected) during a pitch excursion to 45 degrees. The decreasing curve leading up to the 45 degree pitch is due to not initializing the bias filter with enough samples. For these tests, the filter was only initialized with 100 samples, which represents about 15 seconds. However, since the time constant is much larger, the filter should be initialized over more samples. Ideally, the filter should be initialized for τ seconds. The plot still shows some bias effect as well as the effects of undersampling (or slow update rate). Here, the navigation filter gets an update just before the tilt-table stops, but the filter still thinks it has a rate bias and applies it to the next sample, which occurs after the tilt-table has stopped.

Figure 6.5 shows the result of another similar test, only with K1 = 0.1, biasWght = .9999, sampleWght = .0001, and xAccelScale = 4.1. This test re-introduces the low-frequency data (accelometer data) into the integrator. This plot shows yet more accurate performance, but displays an "overshoot" in both directions of the excursion. This overshoot can be alleviated by adjusting the scale factor.
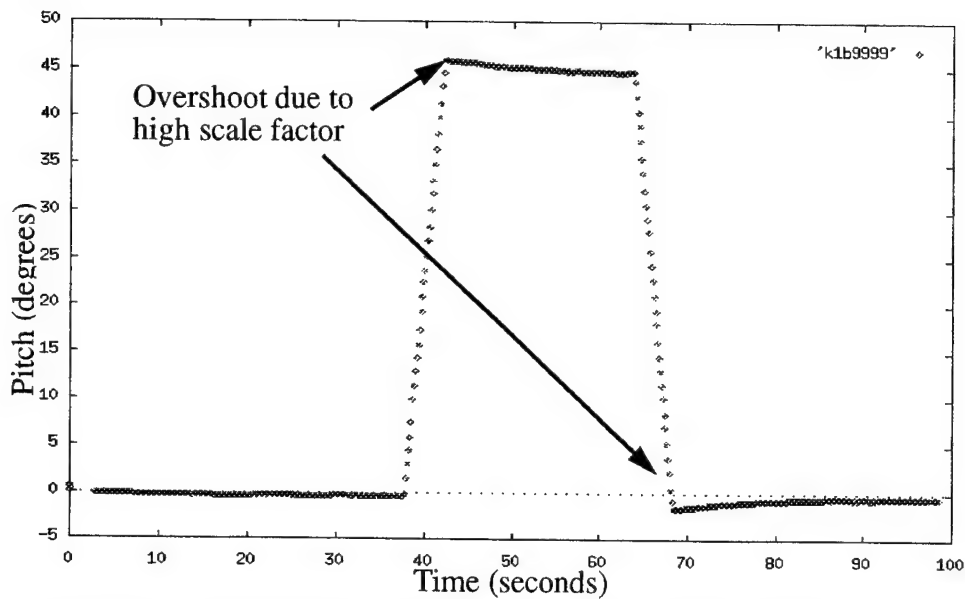
Figure 6.5: 45 Degree Pitch Excursion w/K1=0.1, biasWght=.9999, scale factor= 4.1

Figure 6.6 shows the result of another similar test, only with K1 = 0.1, biasWght = .9999, sampleWght = .0001, and an adjusted scale factor, xAccelScale = 3.895. This plot shows an undershoot, indicating the scale factor was too low. Additionally, it still shows a small amount of sampling effect. This can only be corrected by getting the navigation filter to run faster. This can be easily accomplished by adding a math co-processor to the SANS computer.
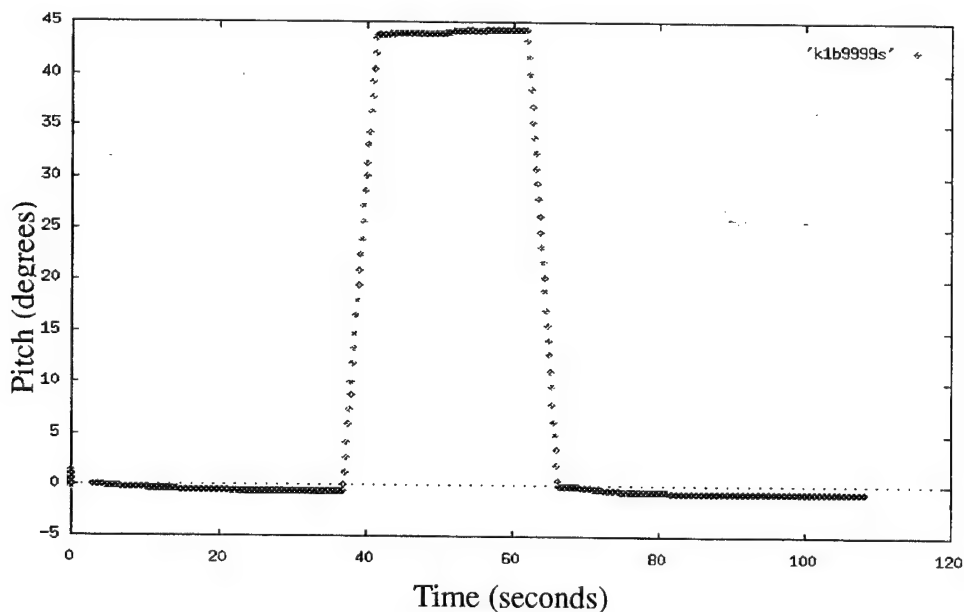
Figure 6.6: 45 Degree Pitch Excursion w/K1=0.1, biasWght=.9999, scale factor= 3.895

48

C. STATIC GPS TEST

The purpose of this test was to ensure proper functioning of the DGPS and the Kalman filter in calculating position. With the SANS stationary, operating, and receiving differentially corrected GPS fixes, position updates were recorded over a 30 minute period at a rate of 10 Hz. Figure 6.7 confirms proper operation of the DGPS hardware and the Kalman filter software. The position updates show an oscillating behavior roughly centered about the origin.
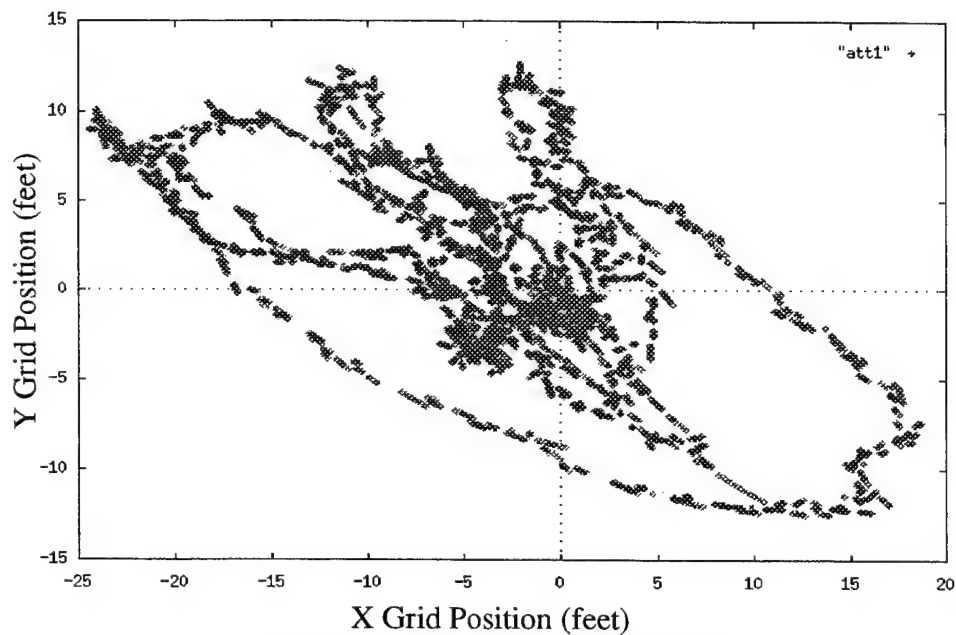
Figure 6.7: Static GPS Test Results

The results shown in Figure 6.7 are well within the 10 meter requirements outlined in [Kwak 93]. By looking at a plot of the x position, it was determined that the greatest drifts shown in Figure 6.7 occurred early into the test, and as time went on, these drifts got smaller. This plot shows how the SANS Kalman filter "learns" as time progresses. In fact, this drift becomes smaller and smaller due to the accumulation of apparent current. At the beginning of the test, the apparent current is small. Any error in location is attributed to apparent current, so as time goes on, the magnitude of the apparent current grows. Because the apparent current velocity is added to the North & East velocity, the difference between the INS determined position and the GPS determined position decreases with time. Figure 6.8 shows a plot of the magnitude of the apparent current during the same test as that depicted in Figure 6.7. This plot shows that the apparent current starts out small, but quickly adjusts to a randomly varying value reflecting the accumulation of all system errors.

49
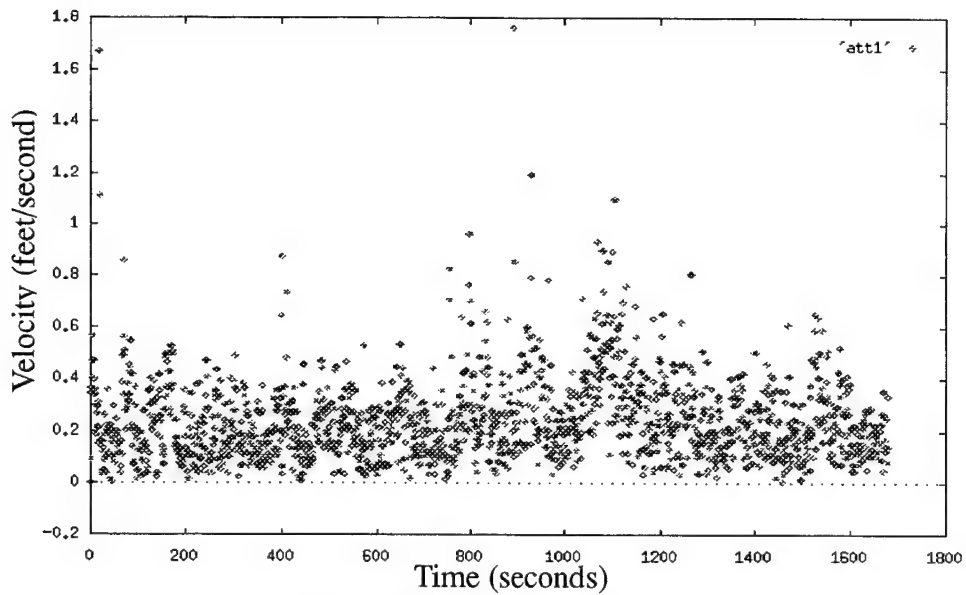
Figure 6.8: Apparent Current

D. STATIC HEADING TEST

The purpose of this test was to ensure that the heading calculated by the Kalman filter was in fact continuous in behavior. That is, a rotation through North in the clockwise direction should not cause the heading value to go back to zero, but rather continue to increase. With the SANS stationary and running, the compass was rotated several complete rotations in both the clockwise and counterclockwise directions. As Figure 6.9 shows, the heading value is in fact continuous in nature as the plot of heading does not show any discontinuity at the zero degree crossing point.
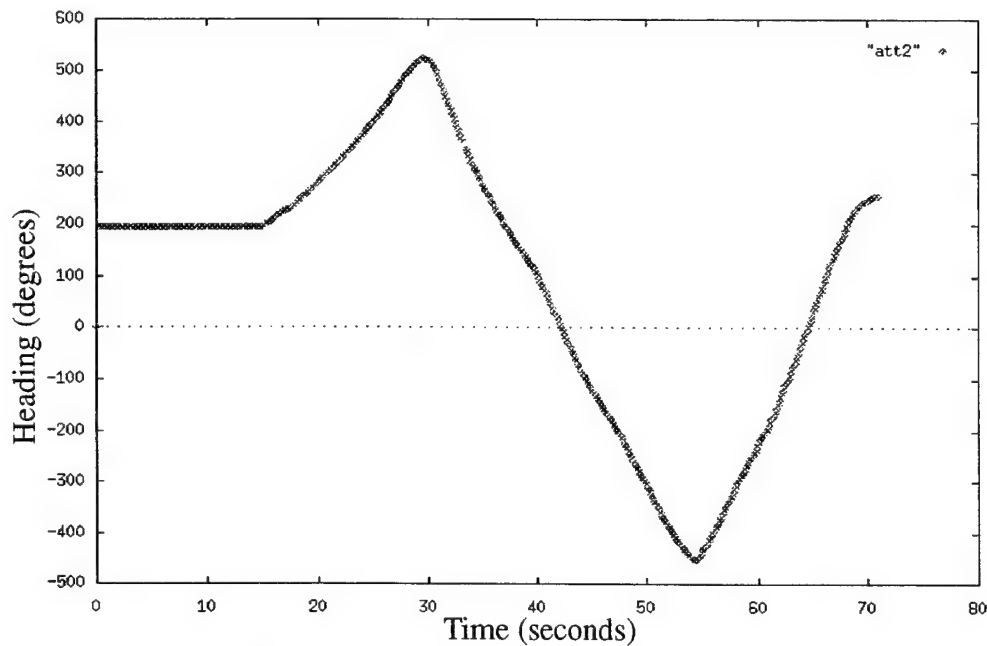
Figure 6.9: Static Heading Test Results

E. SUMMARY

This chapter provides an explanation and results analysis of the experimental tests performed on the SANS to determine its proper functioning under the newly designed architecture.

Dynamic tilt-table tests were performed in order to "tune" the software scale factors, bias values, and gain factor K1 used in the Kalman filter. These test also confirmed the proper operation of the SANS in general.

The DGPS and the associated software was tested by conducting a static test during which the SANS remained stationary while getting position fixes. This test confirmed the proper operation of the DGPS hardware and software. The behavior of the results also indicate that the SANS software Kalman filter is properly attributing its navigation errors to apparent current, as expected.

The compass and associated software was tested by analyzing the heading output of the SANS while the compass was rotated through several complete rotations. The results show a continuous heading output from the SANS, that is, there are no discontinuous "jumps" as the compass rotates through North.

Though extensive testing of the SANS still remains, initial testing indicates qualitatively that the SANS does function properly. Based on observations during this testing, the SANS shows an improved performance over the previous proof of concept prototype. Mainly, the SANS now pro-

51

vides an improved update rate of up to 10 Hz in comparison to the 5 Hz of the prototype. This increase in update rate has reduced the effects of undersampling experienced with the prototype and explained in [Bachmann 95]. Figure 6.6 provides insight to conclude the SANS is still suffering from some effects of undersampling. This effect will most likely be alleviated with the planned addition of a math co-processor to the E.S.P. CPU Module.

VII. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

A. CONCLUSIONS

The research issues addressed by this thesis were 1) Evaluate the hardware architecture of the prototype SANS, 2) Develop a hardware configuration which will enable the SANS to be housed in one small, self-contained package, and 3) Evaluate the feasibility of an AUV accurately navigating from point to point in open-ocean transit using this hardware configuration.

The work conducted in addressing the first of these questions revealed several sources of navigation inaccuracy. The analog low-pass filter circuitry used in the prototype SANS was not adequately attenuating the angular-rate sensor noise below the resolution of the least significant bit of the 12-bit A/D, and was over-designed to require high current, thus generated a great deal of heat. It was also discovered that circuit components were being adversely affected by heat. Specifically, the x-angular-rate signal displayed a significant growth behavior over time as the circuit heated up. Both of these problems were alleviated by employing higher quality, commercially available analog filters, and also employing a switching DC-DC converter in place of the linear regulators employed in the prototype SANS. All this circuitry was designed into a small double-sided PCB.

In addressing the second of these questions, an emphasis on making the SANS integrated and self-contained produced a configuration in which all SANS components are physically packaged in a project box measuring 17.5" X 6.125" X 3.0". In the prototype SANS, the IMU, compass, GPS antenna, and water speed/depth sensors were physically separated from the computer and DGPS receiver. The newly designed SANS no longer requires the use of the data logging computer (which did the A/D conversion and assembled data packets) or the Motorola modems. The SANS now employs a E.S.P. 486 DX2 50 mini computer with an internal 12-bit A/D converter, DC-DC converter module (enables computer to be powered from a 12 Volt battery), Ethernet module (enables SANS to be networked in a client/server environment), PCMCIA module (provides SANS with expandability), and memory storage modules. From observing the SANS operation during bench testing, it is evident the system runs faster (gives faster update rates) since it is no longer hindered by the "bottleneck" induced by the Xmodem transfer of data employed by the prototype SANS.

Though the last research question was not directly addressed in detailed at-sea testing, those tests that were conducted for this thesis qualitatively indicate that the newly designed hardware configuration provides a higher level of performance to the SANS. From this result, it can be extrapolated that the feasibility of an AUV accurately navigating during open-ocean transit, is at least as high as that achieved with the prototype SANS. It is likely that the newly designed SANS will demonstrate increased accuracy in at-sea trials.

B. RECOMMENDATIONS FOR FUTURE WORK

There are many ways to build on the foundation this and related research has established. In addition to comprehensive at-sea testing, and since this project research is dynamic in nature, there are several other areas that remain to be addressed as they pertain to the SANS. Before any worthwhile at-sea testing can be accomplished, further tilt-table/bench testing needs to be conducted. In order to optimize the performance of the Kalman filter used for inertial navigation, testing needs to be done in order to better optimize the gains in the Kalman filter. By analyzing the data from varied accelerometer and angular-rate tests, one can better choose what these gain values need to be.

An issue directly related to testing is that of post-processing. The post-processing code written to run on the previous prototype SANS should be updated/changed to enable its use on the newly designed SANS. Through post-processing test data, one can more easily run and re-run the tests in the goal of more easily optimizing the Kalman filter gains. The current version of the SANS software does not save the "raw" sensor data. This should be made possible. With the help of post-processing code, one needs only run a particular tilt/bench/cart test one time, and then take the raw data back, analyze it, adjust gains and re-run the test in order to optimize gains.

There are hardware issues that remain and are areas for future work. The SANS uses a paddle-wheel type water speed sensor. This is a rather crude and inaccurate sensor, and should be upgraded to one which can deliver reliable data at the slower velocities under which the towfish and Phoenix AUV operate. Additionally, in order to conduct future at-sea testing, the DGPS antenna currently used on the SANS will have to be upgraded. In previous sea trials, the towfish would become fowled by seaweed. Under these conditions, the current DGPS antenna will not survive. Due to difficulties in acquiring software drivers for the Ethernet module, the network connection between the SANS and an external PC was not established during the course of this

thesis. In order to conduct at-sea testing, this network connection is a must, so work still remains in bringing this aspect of the SANS to fruition.

Lastly, the final step in this research will be incorporation of the SANS into the NPS AUV. In this work, there should not be any significant changes to the hardware. However, there will be requirements in interfacing this hardware with the operating system running the NPS AUV. It may be beneficial from both a space-saving aspect as well as from the aspect of having more computing power, to transfer all software operation over to the Sun Voyager workstation which currently accomplishes mission control for the NPS AUV. In this case, the issues of serial port communications, and A/D conversion will have to be readdressed before this can be successful. On the other hand, there is a distinct advantage to leaving the SANS as is and simply integrating it into the NPS AUV as another client in its established client/server environment. A detailed study of these alternatives will be required before the best solution can be developed.

APPENDIX A: Real Time Navigation Source Code(C++)

A. TOWTYPES.H

```
#ifndef __TOETYPES_H
#define __TOETYPES_H

#include <stdio.h>
#include <dos.h>
#include <time.h>

#include "globals.h"    // Types used by serial communications software

#define GPSBLOCKSIZE 76// Size of Motorola @@Ea position message
#define COMPSIZE 60     // Max size of compass message
#define biasNumber 10   // Number of packets used to calculate initial bias

#define ONE_G 32.2185// One g in feet per second
#define GRAVITY 32.2185 // In feet per second

#define TicksToSecs(x)  ((double) ((10 * x) / 182))

typedef char   ONEBYTE;
typedef short  TWOBYTE;
typedef long   FOURBYTE;

typedef unsigned char   UNSIGNED_ONEBYTE;
typedef unsigned short  UNSIGNED_TWOBYTE;
typedef unsigned long   UNSIGNED_FOURBYTE;

// Holds lat/long expressed in milseconds
struct latLongMilSec {
     long latitude;
     long longitude;
};

// Holds a latitude or longitude expressed in hours minutes and degrees
struct T_GEODETIC {
  TWOBYTE            degrees;
  UNSIGNED_TWOBYTE   minutes;
  double             seconds;
};

// Holds a latitude and longitude expressed as T_GEODETICs
struct latLongPosition {
  T_GEODETIC latitude;
  T_GEODETIC longitude;
};

// Holds a grid position
struct grid {
   double x,y,z;
};
```

```c
// 3 X 3 matrix
struct matrix {
    float element[3][3];
};

// 3 X 1 matrix or vector
struct vector {
    float element[3];
};

// Oversize area to hold a GPS message
typedef BYTE GPSdata[2 * GPSBLOCKSIZE];

// Defines a type for holding compass messages
typedef BYTE compData[2 * COMPSIZE];

// Structure for passing around various types of INS information.
// The positions in the sample field of a stampedSample structure
    // sample[0]: x acceleration
    // sample[1]: y acceleration
    // sample[2]: z acceleration
    // sample[3]: phi
    // sample[4]: theta
    // sample[5]: psi
    // sample[6]: water speed
    // sample[7]: heading

struct stampedSample {
    grid est;                 //position as estimated by the INS.
    float rawSample[8];   //Original readings for post processing
    double sample[8];     //sampler converted sample.
    float deltaT;
    float current;
};
#endif
```

B. LOCATION.H

```c
//Conversion constants for location of
//36:35:42.2N and121:52:28.7W
#define LatToFt 0.10134 //converts degrees Latitude to ft
#define LongToFt 0.08156 //converts degrees Longitude to ft
#define HemisphereConversion -1   //-1 if west of of Greenwich

#define RADIANMAGVAR 0.261799   // Local area Magnetic variation in radians

#define xyAccelLimit ONE_G// Max accell in x and y diretion
#define zAccelLimit 2 * ONE_G   // Max accel in z direction
#define rateLimit  0.872665// Max rotational rate in radians
#define speedLimit 25.3 //Max water speed
#define headingLimit 2 * M_PI

#define pScale 1.0 //roll
```

```
#define qScale 1.0 //pitch
#define rScale 1.0 //yaw

#define xAccelScale 1.34   //1.078 //pitch
#define yAccelScale 3.895 //roll
#define zAccelScale 1.34   //1.038

#define pUnits(angular) (pScale * (((angular-2047.0) / 2047.0 )
                         * 50.0) * (M_PI/180.0))
#define qUnits(angular) (qScale * (((angular-2047.0) / 2047.0 )
                         * 50.0) * (M_PI/180.0))
#define rUnits(angular) (rScale * (((angular-2047.0) / 2047.0 )
                         * 50.0) * (M_PI/180.0))

#define xAccelUnits(linear) (xAccelScale * ((linear-2047.0) /
                             2047.0 ) * GRAVITY)
#define yAccelUnits(linear) (yAccelScale * ((linear-2047.0) /
                             2047.0 ) * GRAVITY)
#define zAccelUnits(linear) (zAccelScale * ((linear-2047.0) /
                             2047.0) * (2.0 * GRAVITY))
#define waterSpeedScale 1.0 //1.827
#define depthUnits(depth) (((depth - 819.0) / (4095.0-819.0))
                            * 180.0)
#define waterSpeedUnits(speed)  (waterSpeedScale * ((speed -
                    2047.0) / 2048.0) * 25.3) //feet per second

#define radToDeg(180.0/M_PI)
#define degToRad  (M_PI/180.0)
```

C. TOWFISH.CPP

```cpp
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <conio.h>

#include "toetypes.h"
#include "nav.h"

int   breakHandler(void);
void screenSetUp(void);
void printPosition (const latLongPosition&);

/***********************************************************************
PROGRAM:   Main
AUTHOR:    Eric Bachmann, Dave Gay
DATE:      11 July 1995
FUNCTION: Drives the navigator and its associated software. Counts
          the positions and displays each to the screen. Exited only when
          control break is entered at the keyboard.
RETURNS:   0
CALLED BY:none
CALLS:     initializeNavigator (nav.h)
           navPosit (nav.h)
           printPosition
           breakHandler
***********************************************************************/

int
main (int argc, char *argv[])
{
   ctrlbrk(breakHandler);  // trap all breaks to release com ports
   setcbrk(1);    // turn break checking on at all times

   char *dataFile, *scriptFile, *attitudeFile;

   switch (argc) {
      case 2:
         scriptFile = new char[strlen(argv[1])];
         strcpy(scriptFile, argv[1]);//explicit script file only
         dataFile = "data";//default raw data file
         attitudeFile = "attitude";
         break;

      case 3:
         scriptFile = new char[strlen(argv[1])];
         strcpy(scriptFile, argv[1]);//explicit script file
         dataFile = new char[strlen(argv[2])];
         strcpy(dataFile, argv[2]);//explicit data file
         attitudeFile = "attitude";
         break;
```

```cpp
      case 4:
         scriptFile = new char[strlen(argv[1])];
         strcpy(scriptFile, argv[1]);//explicit script file
         dataFile = new char[strlen(argv[2])];
         strcpy(dataFile, argv[2]);//explicit data file
         attitudeFile = new char[strlen(argv[3])];
         strcpy(attitudeFile, argv[3]);//explicit attitude file
         break;

      default:
         scriptFile = "script";//default script file
         dataFile = "rawdata";//default raw data file
         attitudeFile = "attitude";
}

   clrscr();

   cout << "\nWriting script information to " << scriptFile;
   cout << "\nWriting binary data to " << dataFile;
   cout << "\nWriting attitude data to " << attitudeFile << endl;

   cout << "\nInitializing . . .";

   //Instantiate the navigator
   navigator nav1(scriptFile, dataFile, attitudeFile);

   latLongPosition currentLocation; // Lat/Long of most recent fix
   Boolean  fixReceived = FALSE;//True if a new fix was recieved
   int      fixCount=0;   // Count of navigation fixes recieved

   //Initialize the navigator
   currentLocation = nav1.initializeNavigator();

   gotoxy(1,6);
   cout << "Initialization Complete!\n";
   cout << "Initial Position:\n";

   //Print the initial position
   cout << "latitude: " << currentLocation.latitude.degrees << ':'
      << currentLocation.latitude.minutes << ':'
      << currentLocation.latitude.seconds << endl;
   cout << "longitude: " << currentLocation.longitude.degrees << ':'
       << currentLocation.longitude.minutes << ':'
       << currentLocation.longitude.seconds;

   screenSetUp();

   while (TRUE)  {
      // Attempt to get a fix from the navigator
      fixReceived = nav1.navPosit(currentLocation)
      if (fixReceived) {
         // New fix recieved
         gotoxy(8,11);
         cout << ++fixCount;
```

61

```
            printPosition(currentLocation);
        }
    }
}

/******************************************************************
PROGRAM:    printPositon
AUTHOR:     Eric Bachmann, Dave Gay
DATE:       11 July 1995
FUNCTION:   Displays position to the screen
RETURNS:    void
CALLED BY: mail
CALLS:      none
******************************************************************/

void
printPosition (const latLongPosition& posit)
{
    gotoxy(11,14);
    cout << posit.latitude.degrees << ':'
        << posit.latitude.minutes << ':' << posit.latitude.seconds << endl;
    gotoxy(12,15);
    cout << posit.longitude.degrees << ':'
        << posit.longitude.minutes << ':' << posit.longitude.seconds << endl;
}


/******************************************************************
PROGRAM:    breakHandler
AUTHOR:     Eric Bachmann, Dave Gay
DATE:       11 July 1995
FUNCTION:   Cleans up com ports upon program exit.
RETURNS:    0
CALLED BY: main
CALLS:      cleanup (portBank.h)
******************************************************************/

int
breakHandler(void)
{
    COMports.cleanup();
    exit(0);
    return 0;        // keep the compiler happy
}
```

```
/**********************************************************************
PROGRAM:screenSetup
AUTHOR:Eric Bachmann, Randy Walker
DATE:12 May 1996
FUNCTION:Sets up the output screen
RETURNS:0
CALLED BY:  main
CALLS:      none
**********************************************************************/
void
screenSetUp(void)
{
   gotoxy(4,11);
   cout << "Fix ";

   gotoxy(1,14);
   cout << "Latitude: " << "\nLongitude: ";

   gotoxy(1,17);
   cout << "Roll: " << "\nPitch: ";

   gotoxy(1,25);
   cout << "deltaT: ";

   int col(45),row(1);

   gotoxy(col,row++);
   cout << "x accel: ";
   gotoxy(col,row++);
   cout << "y accel: ";
   gotoxy(col,row++);
   cout << "z accel: ";
   gotoxy(col,row++);
   cout << "phi dot: ";
   gotoxy(col,row++);
   cout << "theta dot: ";
   gotoxy(col,row++);
   cout << "psi dot: ";
   gotoxy(col,row++);
   cout << "water speed: ";
   gotoxy(col,row++);
   cout << "heading: ";

   col = 45;
   row = 12;

   gotoxy(col,row++);
   cout << "x: ";
   gotoxy(col,row++);
   cout << "y: ";
   gotoxy(col,row++);
   cout << "z: ";
   gotoxy(col,row++);
   cout << "phi: ";
```

```
        gotoxy(col,row++);
        cout << "theta: ";
        gotoxy(col,row++);
        cout << "psi: ";

        gotoxy(45,20);
        cout << "Bias Values";

        gotoxy(60,20);
        cout << "Current Values";

}
```

D. NAV.H

```
#ifndef __NAVIGATOR_H
#define __NAVIGATOR_H
#include <stdio.h>
#include <fstream.h>
#include <iostream.h>
#include <math.h>

#include "toetypes.h"
#include "gps.h"
#include "ins.h"
#include "location.h"

// Converts milseconds to degrees
#define MSECS_TO_DEGREES (1.0/(1000.0 * 3600.0))

/*********************************************************************
    CLASS:      navigator
    AUTHOR:     Eric Bachmann, Dave Gay
    DATE:       11 July 1995
    FUNCTION:   Combines GPS and INS information to return the current
                estimated position.
*********************************************************************/

class navigator {

  public:

    //Constructor, opens script and data files
    navigator(char scriptFile[] = "navScript",char dataFile[] = "navData",
    char attitudeFile[] = "navatt"): positionData (scriptFile),
    attitudeData(attitudeFile),elapsedTime(0.0), fixCount(0), gpsSpeedSum(0.0),
    insSpeedSum(0.0)
    {if ((rawData = fopen(dataFile, "wb")) == NULL) {
        cout << "NO RAW DATA RECORD";}
    }

    //Destructor, closes script and data files
    ~navigator() {positionData.close();attitudeData.close();fclose(rawData);}
```

```cpp
    //provides the navigator's best estimate of current position
    Boolean navPosit (latLongPosition&);

    //Initialize the navigator
    latLongPosition initializeNavigator();

  private:

    float elapsedTime; // Tracks time since navigator was initialized
    int fixCount;       // the number of position fixes obtained

    double gpsSpeed, insSpeed;
    double gpsSpeedSum, insSpeedSum;

    INS ins1;//INS object instance.
    GPS gps1;//GPS object instance.

    ofstream attitudeData;// Post processing attitude data.
    ofstream positionData;  // Position script file
    FILE *rawData;// Post processing binary data file.

    latLongMilSec origin; //lat-long of navigational origin

    //Write position information to script file
    void writeScriptPosit(int, latLongMilSec&, char);

    //Write an INS packet and its timeStamp to the outPut file
    void writeInsData(const stampedSample& drPosition);

    //Write a GPS message to the outPut file.
    void writeGpsData(const GPSdata& satPosition);

    //Returns the position in Miliseconds
    latLongMilSec getMilSec(const GPSdata&);

    //Convert position in milSec to degress, minutes, seconds and milsec
    latLongPosition milSecToLatLong(const latLongMilSec&);

    //Convert xy (grid) position to lat long
    latLongMilSec gridToMilSec(const grid&);

    //Converts lat/long to xy position
    grid milSecToGrid(const latLongMilSec&);

    //Parses and returns the time of a GPS message.
    double getGpsTime(const GPSdata& rawMessage);

    //Parses and returns the velocity in fps of a GPS message.
    double getGpsVelocity(const GPSdata& rawMessage);
};
#endif
```

E. NAV.CPP

```cpp
#include "nav.h"
#include "signal.h"
#define SIGFPE 8// Floating point exception

/************************************************************************
PROGRAM:  navPosit
AUTHOR:   Eric Bachmann, Dave Gay
DATE:     11 July 1995
FUNCTION:Provides the navigator's best estimate of current position.
   Attempts to obtain GPS and INS position fixes from the gps
   and ins objects and copies the most accurate fix available
   into the input argument 'navPosition'. Writes the raw position
   fix data to the output file for post processing. Sets a return
   flag to indicate whether a valid fix was obtained.
RETURNS:  TRUE, a valid position fix is in the variable 'navPosition'.
          FALSE, otherwise.
CALLED BY:towfish.cpp (main)
CALLS:        gpsPosition (gps.h)
              correctPosition (ins.h)
              insPosition (ins.h)
              getMilSec (nav.h)
              gridToMilSec (nav.h)
              milSecToGrid (nav.h)
              milSecToLatLong (nav.h)
              writeScriptPosit (nav.h)
************************************************************************/
void fpeNavPosit(int sig)
{if (sig == SIGFPE) cerr << "floating point error in navPosit\n";}


Boolean
navigator::navPosit (latLongPosition& navPosition)
{

   signal (SIGFPE, fpeNavPosit);
   GPSdata satPosition;       // the latest GPS position
   stampedSample drPosition; // the latest INS position
   latLongMilSec gpsMilSec;  // the latest GPS position in milseconds
   latLongMilSec insMilSec;  // the latest INS position in milseconds

   //Attempt to get the INS and GPS positions
   Boolean insFlag = ins1.insPosition(drPosition);
   Boolean gpsFlag = gps1.gpsPosition(satPosition);

   //INS and GPS positions obtained?
   if (insFlag && gpsFlag) {
      gotoxy(20,11);
      cout << "GPS";
      // Write INS packet and attitude info to an output file
      elapsedTime += drPosition.deltaT;
      writeInsData(drPosition);
      //Write GPS message to output file.
      writeGpsData(satPosition);
```

```
    //Parse position from GPS messsage
    gpsMilSec = getMilSec(satPosition);
    //Write milsec position to script file
    writeScriptPosit(++fixCount, gpsMilSec, 'G');
    //Pass GPS position to INS object for navigation corrections.
    ins1.correctPosition(milSecToGrid(gpsMilSec), getGpsTime(satPosition));
    //Covert position in mil sec to latitude and longitude.
    navPosition = milSecToLatLong(gpsMilSec);
    return TRUE;
}
else {
    //Only INS position obtained?
    if (insFlag) {
        gotoxy(20,11);
        cout << "    ";
        // Write INS Packet to output file.
        elapsedTime += drPosition.deltaT;
        writeInsData(drPosition);
        insMilSec = gridToMilSec(drPosition.est);
        //Write milsec position to script file
        writeScriptPosit(++fixCount, insMilSec, 'I');
        navPosition = milSecToLatLong(insMilSec);

        insSpeed = drPosition.sample[6];

        return TRUE;
    }
    else {
        // Only GPS position obtained?
        if (gpsFlag) {
            gotoxy(20,11);
            cout << "GPS";
            // Write GPS message to output file.
            writeGpsData(satPosition);
            //Parse position from GPS messsage
            gpsMilSec = getMilSec(satPosition);
            //Write milsec position to script file
            writeScriptPosit(++fixCount, gpsMilSec, 'G');
            //Pass GPS position to INS object for navigation corrections.
            ins1.correctPosition(milSecToGrid(gpsMilSec),
                                            getGpsTime(satPosition));
            //Convert position in mil sec to lat/long.
            navPosition = milSecToLatLong(getMilSec(satPosition));

            gpsSpeed = getGpsVelocity(satPosition);
            gpsSpeedSum += gpsSpeed;
            insSpeedSum += insSpeed;
            gotoxy(58,9);
            cout << gpsSpeed;
            gotoxy(58,10);
            cout << gpsSpeedSum / insSpeedSum;

            return TRUE;
        }
```

67

```
        else {
          return FALSE; // No new position available
        }
      }
   }
}
/*************************************************************************
PROGRAM:    writeScriptPosit
AUTHOR:     Eric Bachmann, Dave Gay
DATE:       11 July 1995
FUNCTION:   Writes the fix number, the position in milSec and the type
            of fix to the script file.
RETURNS:    void
CALLED BY:  navPosit (nav.cpp)
            initialPosit  (nav.cpp)
CALLS:      None
*************************************************************************/

void
navigator::writeScriptPosit(int fixNumber, latLongMilSec& posit, char fixType)
{
   positionData << fixNumber << ' '
                << posit.latitude << ' '
                << posit.longitude << ' '
                << fixType << ' '
                << elapsedTime << endl;
}

/*************************************************************************
PROGRAM:    writeInsData
AUTHOR:     Eric Bachmann, Dave Gay
DATE:       11 July 1995
FUNCTION:   Writes the packet and the time stamp contained in a stamped
            sample to the out put file for post processing.
RETURNS:    void
CALLED BY:  navPosit (nav.cpp)
CALLS:      None
*************************************************************************/

void
navigator::writeInsData(const stampedSample& drPosition)
{

//MUST ADD CODE TO RECORD DATA FOR POST PROCESSING HERE

   //Output attitude data to a file
   attitudeData << elapsedTime << ' '
                << drPosition.sample[0] << ' '
                << -1.0 * drPosition.sample[1] << ' '
                << drPosition.sample[2] << ' '
                << (radToDeg * drPosition.sample[3]) << ' '
                << (radToDeg * drPosition.sample[4]) << ' '
                << (radToDeg * drPosition.sample[5]) << ' '
                << drPosition.sample[6] << ' '
```

```
                      << (radToDeg * drPosition.sample[7]) << ' '
                      << drPosition.current << endl;

}
/*******************************************************************
PROGRAM:      writeGpsData
AUTHOR:       Eric Bachmann, Dave Gay
DATE:         11 July 1995
FUNCTION:     Writes a raw GPS message to a binary output file for
              post processing.
RETURNS:      void
CALLED BY:    navPosit (nav.cpp)
CALLS:        None
*******************************************************************/

void
navigator::writeGpsData(const GPSdata& satPosition)
{
    for( int j = 0; j < GPSBLOCKSIZE; j++) {
        putc(satPosition[j], rawData);
    }
}


/*******************************************************************
PROGRAM:      initializeNavigator
AUTHOR:       Eric Bachmann, Dave Gay
DATE:         11 July 1995
FUNCTION:     Obtains an initial GPS fix for use as a navigational origin for
    grid positions used by the INS object. Saves the origin and passes
    it to the INS object in latLong form.

    RETURNS:  TRUE
CALLED BY:    towfish (main)
CALLS:        gpsPosition (gps.cpp)
              correctPosition (ins.cpp)
              getMilSec (nav.cpp)
              milSecToGrid (nav.cpp)
*******************************************************************/

latLongPosition
navigator::initializeNavigator()
{
    stampedSample biasPackets[biasNumber];

    GPSdata satPosition;    //gps position message

    // Loop until an initial GPS fix is obtained.
    while(!gps1.gpsPosition(satPosition)) { /* */ }

    // Write GPS message for the grid origin to output file.
    writeGpsData(satPosition);
    //Save navigational origin for later grid position conversions.
    origin = getMilSec(satPosition);
    //Write the initial position to the script file
```

69

```
   writeScriptPosit(0, origin, 'G');
   //Pass time of first GPS fix to INS object initialization routine.
   ins1.insSetUp(getGpsTime(satPosition), biasPackets);

   for (int i = 0; i < biasNumber; i++) {
      writeInsData(biasPackets[i]);
   }
   insSpeed = biasPackets[biasNumber - 1].sample[6];

   gpsSpeed = getGpsVelocity(satPosition);

   //Return the initial position to the caller.
   return milSecToLatLong(origin);
}


/*************************************************************************
PROGRAM:getMilSec
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Extracts a position in mili seconds from a Motorola (@@Ba)
   position contained in the input argument 'rawMessage' and returns it.
RETURNS:The latitude and longitude in milseconds.
CALLED BY:   navPosit (nav.cpp)
             initializeNavigator (nav.cpp)
CALLS:none.
*************************************************************************/

latLongMilSec
navigator::getMilSec(const GPSdata& rawMessage) {

   FOURBYTE temps4byte;
   latLongMilSec position;

   temps4byte        = rawMessage[15];
   temps4byte        = (temps4byte<<8) + rawMessage[16];
   temps4byte        = (temps4byte<<8) + rawMessage[17];
   temps4byte        = (temps4byte<<8) + rawMessage[18];

   position.latitude = temps4byte;

   temps4byte        = rawMessage[19];
   temps4byte        = (temps4byte<<8) + rawMessage[20];
   temps4byte        = (temps4byte<<8) + rawMessage[21];
   temps4byte        = (temps4byte<<8) + rawMessage[22];

   position.longitude = temps4byte;

   return position;
}
```

```
/*********************************************************************
PROGRAM:milSecToLatLong
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Converts a position expressed in totally in mili seconds to
   degress, minutes, seconds and mili seconds and returns the result.
RETURNS:The position in degress, minutes, seconds and mili seconds.
CALLED BY:   navPosit (nav.cpp)
CALLS:none
*********************************************************************/

latLongPosition
navigator::milSecToLatLong(const latLongMilSec& milSec) {

   latLongPosition   position;
   double degrees, minutes;
   degrees = (double)milSec.latitude * MSECS_TO_DEGREES;
   position.latitude.degrees = (TWOBYTE)degrees;

   if(degrees < 0)
      degrees = fabs(degrees);

   minutes = (degrees - (TWOBYTE)degrees) * 60.0;
   position.latitude.minutes = (TWOBYTE)minutes;
   position.latitude.seconds = (minutes - (TWOBYTE)minutes) * 60.0;

   degrees = (double)milSec.longitude * MSECS_TO_DEGREES;
   position.longitude.degrees = (TWOBYTE)degrees;

   if(degrees < 0)
      degrees = fabs(degrees);

   minutes = (degrees - (TWOBYTE)degrees) * 60.0;
   position.longitude.minutes = (TWOBYTE)minutes;
   position.longitude.seconds = (minutes - (TWOBYTE)minutes) * 60.0;

   return position;

}
```

```
/************************************************************************
PROGRAM:gridToMilSec
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Convert a grid position to a latitude and longitude in mil-
   seconds and returns the result.
RETURNS:The latitude and longitude in milseconds.
CALLED BY:navPosit (nav.cpp)
CALLS:none
************************************************************************/
void fpeGridToMilSec(int sig)
{if (sig == SIGFPE) cerr << "floating point error in gridToMilSec\n";}

latLongMilSec
navigator::gridToMilSec(const grid& posit)
{
    signal(SIGFPE, fpeGridToMilSec);
    latLongMilSec  latLong;

    //converts grid in ft to latitude
    latLong.latitude = origin.latitude + (posit.x / LatToFt);
    //converts grid in ft to longitude
    latLong.longitude = origin.longitude +
        HemisphereConversion * (posit.y / LongToFt);
    return latLong;
}


/************************************************************************
PROGRAM:milSecToGrid
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Convert a latitude and longitude expressed in milseconds to
   a grid position based on the lat/long of the grid origin.
RETURNS:The grid position
CALLED BY:navPosit (nav.cpp)
          initializeNavigator (nav.cpp)
CALLS:none
COMMENTS:altitude is always assumed to be zero.

************************************************************************/

//Converts latitude/longitude to xy coords in ft from origin
grid
navigator::milSecToGrid(const latLongMilSec& posit)
{
    grid  position;

    position.x = (posit.latitude - origin.latitude) * LatToFt;
    position.y = HemisphereConversion *
        (posit.longitude - origin.longitude) * LongToFt;
    position.z = 0;

    return position;
}
```

```
/**********************************************************************
PROGRAM:getGpsTime
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Parse the time of a gps message.
RETURNS:The time of the gps message in seconds
CALLED BY:navPosit (nav.cpp)
         initializeNavigator (nav.cpp)
CALLS:none
**********************************************************************/
double
navigator::getGpsTime(const GPSdata& rawMessage)
{
    UNSIGNED_ONEBYTE      tempchar, hours, minutes;
    UNSIGNED_FOURBYTE     tempu4byte;
    double seconds;

    hours   = rawMessage[8];
    minutes = rawMessage[9];

    tempchar          = rawMessage[10];
    tempu4byte        = rawMessage[11];
    tempu4byte        = (tempu4byte<<8) + rawMessage[12];
    tempu4byte        = (tempu4byte<<8) + rawMessage[13];
    tempu4byte        = (tempu4byte<<8) + rawMessage[14];
    seconds = (double)tempchar + (((double)tempu4byte)/1.0E+9);

    return hours * 3600.0 + minutes * 60.0 + seconds;
}

/**********************************************************************
PROGRAM:getGpsVelocity
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Parse the velocity out of a gps message.
RETURNS:The velocitiy of the gps message in feet per second
CALLED BY:navPosit (nav.cpp)
         initializeNavigator (nav.cpp)

CALLS:none
**********************************************************************/

double
navigator::getGpsVelocity(const GPSdata& rawMessage)
{
    UNSIGNED_ONEBYTE tempchar=rawMessage[31];

    return (double)(3.2804 * ((tempchar << 8) + rawMessage[32]) / 100.00);
}
```

F. GPS.H

```
#ifndef _GPS_H
#define _GPS_H

#include <iostream.h>
#include <fstream.h>
#include <conio.h>

#include "portbank.h"
#include "toetypes.h"
#include "gpsbuff.h"

/***********************************************************************
CLASS:gps
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
MODIFIED: 15 May 1996 by Eric Bachmann & Randy Walker
FUNCTION:Reads GPS messages from the GPS buffer. Checks for valid
         checksum and minimun number of satelites in view.

***********************************************************************/

class GPS {

  public:

    //Class Constructor
    GPS() : port1(COMports.Init(COM1, BYTE(4), b9600,
             NOPARITY, BYTE(8), BYTE(1), XON_XOFF, messages)) { }

    //returns the latest gps position and a flag
    Boolean gpsPosition (GPSdata&);

  private:

    //buffer for gps data
    GPSbuffer messages;

    //instantiates serial port communications on comm2
    bufferedSerialPort& port1;

    //calculates the check sum of the message
    Boolean checkSumCheck(const GPSdata);

};
#endif
```

G. GPS.CPP

```cpp
#include <math.h>
#include "gps.h"

/*********************************************************************
NAME:gpsPosition
AUTHOR:  Eric Bachmann, Dave Gay
DATE:11 July 1995
MODIFIED: 15 May 1995 BY Eric Bachmann & Randy Walker
FUNCTION:Determines if an updated gps position message is available and
   copies it into the input argument 'rawMessage'. If the message
   has a valid checksum, was obtained with at least three
   satelites in view, and contained the differential correction,  a 'TRUE'
   is returned to the caller,indicating that the message is valid.
RETURNS: TRUE, if a valid position message is contained in the
         input argument.
CALLED BY: navPosit (navigator.h)
CALLS:Get (buffer.h)
          checkSumCheck (gps.h)
*********************************************************************/

Boolean
GPS::gpsPosition (GPSdata& rawMessage)
{
    Boolean checkSumFlag;
    unsigned long Mask(4);

    if (messages.Get(rawMessage)) {

       // Check for a valid check sum and more the 3 satelites and DGPS
       return Boolean((checkSumCheck(rawMessage)) && (rawMessage[39] > 3)
          && ((rawMessage[GPSBLOCKSIZE - 4] & Mask) == Mask));

    }
    else {
       return FALSE;  // No updated position is available.
    }

}
```

```
/**********************************************************************
PROGRAM:checkSumCheck
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
MODIFIED: 15 May, 1996 by Eric Bachmann & Randy Walker
FUNCTION:Takes an exclusive or of bytes 2 through 78 in a Motorola format
         (@@EA) position message and compares it to the checksum of the
         message.
RETURNS: TRUE, if the message contains a valid checksum
CALLED BY:   gpsPosition (gps)
CALLS:       none
**********************************************************************/

Boolean
GPS::checkSumCheck(const GPSdata newMessage)
{
    BYTE chkSum(0);

    for (int i = 2; i < GPSBLOCKSIZE - 3; i++) {
      chkSum ^= newMessage[i];
    }
    return Boolean(chkSum == newMessage[GPSBLOCKSIZE - 3]);
}
```

H. INS.H

```
#ifndef _INS_H
#define _INS_H

#include <time.h>
#include <math.h>
#include <dos.h>
#include <stdio.h>  .
#include <conio.h>
#include <fstream.h>
#include <iostream.h>

#include "toetypes.h"
#include "location.h"
#include "sampler.h"

/**********************************************************************
CLASS:ins
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Takes in linear accelerations, angular rates, speed and
   heading information and uses kalman filtering techniques to return
   a dead reconing position.
**********************************************************************/

class INS {

   public:
```

```
        //Constructor initializes gains
        INS();

        ~INS() {}

        //returns the ins estimated position
        Boolean insPosition(stampedSample&);

        //Updates the x, y and z of the vehicle posture
        void correctPosition(const grid&, double);

        //Sets posture to the origin and developes initial biases
        void insSetUp(double, stampedSample*);

private:

        double posture[6];      // ins estimated posture (x y z phi theta psi)
        double velocities[6];   // ins estimated linear and angular velocities
                                // x-dot y-dot z-dot phi-dot theta-dot psi-dot
        double current[3];      // ins estimated error current
                                // (x-dot y-dot z-dot)

        double lastGPStime;     //time of last gps position fix

        sampler sam1;                //sampler instance

        matrix rotationMatrix;      //body to euler transformation matrix

        double biasCorrection[8];//Software bias corrections for IMU rate
                              //sensors

        // Kalman filter gains.
        float Kone1, Kone2, Ktwo, Kthree1, Kthree2, Kfour1, Kfour2;

        // Finds the difference between two times of struct time type
        double findDeltaT (struct time& next, struct time& last);

        // Transforms from body coordinates to earth coordinates
        // and removes the gravity component
        void transformAccels (double[]);

        // Transforms water speed reading to x and y components
        void transformWaterSpeed (double, double[]);

        // Tranforms body euler rates to earth euler rates.
        void transformBodyRates (double[]);

        // Euler integrates the accelerations and updates the velocities
        void updateVelocities (stampedSample&);

        // Euler integrates the velocities and update the posture
        void updatePosture (stampedSample&);
```

```
        // Builds the body to euler rate matrix
        matrix buildBodyRateMatrix();

        // Builds the body to earth rotation matrix
        void buildRotationMatrix();

        //Calculates the imu bias correction during set up
        void calculateBiasCorrections(stampedSample*);

        //Applies bias corrections to a sample
        void applyBiasCorrections(double sample[]);

};


// Post multiply a matrix times a vector and return result.
vector operator* (matrix&, double[]);

#endif
```

I. INS.CPP

```cpp
#include <iostream.h>
#include "signal.h"
#include "ins.h"
#define SIGFPE 8// Floating point exception

/**********************************************************************
PROGRAM: ins (constructor)
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Constructor initializes kalman filter gains and linear and
   angular velocities.
RETURNS:nothing
CALLED BY:   navigator class
CALLS:       none
**********************************************************************/

INS::INS() : Kone1(0.1), Kone2(0.5),
        Ktwo(0.6),
        Kthree1(0.5), Kthree2(0.5),
        Kfour1(0.5), Kfour2(0.5)
{
   velocities[0] = 0.0;// x dot
   velocities[1] = 0.0;// y dot
   velocities[2] = 0.0;// z dot
   velocities[3] = 0.0;// phi dot
   velocities[4] = 0.0;// theta dot
   velocities[5] = 0.0;// psi dot

   //Set posture to straight and level at the origin.
   posture[0] = 0.0;
   posture[1] = 0.0;
```

```cpp
    posture[2] = 0.0;
    posture[3] = 0.0;
    posture[4] = 0.0;
    posture[5] = 0.0;

    // Initialize error current to zero
    current[0] = 0.0;
    current[1] = 0.0;
    current[2] = 0.0;

}

/*********************************************************************
PROGRAM: findDeltaT
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Converts two times stored in the time structure type of
      dos.h into the time in seconds and returns the difference.
RETURNS:difference in seconds between the two input times.
CALLED BY:    insPosit (ins.cpp)
CALLS:        none
*********************************************************************/
double
INS::findDeltaT (struct time& next, struct time& last)
{
    double present, past;

    present = next.ti_hour * 3600.0 + next.ti_min * 60.0
          + next.ti_sec + next.ti_hund / 100.0;
    past = last.ti_hour * 3600.0 + last.ti_min * 60.0
         + last.ti_sec + last.ti_hund / 100.0;

    // Did 2400 occur between present and past?
    if (present < past) {
      present += 86400.0;
    }

    return present - past;
}
```

```
/********************************************************************
PROGRAM: insPosit
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:    Make dead reckoning position estimation using kalman
   filtering. Inputs are linear accelerations, angular rates, speed and
   heading.  Primary input data is obtained from a sampler object via the
   getSample method. This data is stored in the sample filed of a
   stampedSample structure called newSample. The sample field is then
   used as a working variable as the linear accelerations and angular
   rates it contains are converted to earth coordinates and integrated
   to determine current velocities and posture. The data is complimentary
   filtered against itself, speed and magnetic heading.
RETURNS:position in grid coordinates as estimated by the INS
CALLED BY:   navPosit (nav.cpp)
CALLS:       getSample (sampler.cpp)
             findDeltaT (ins.cpp)
             transformBodyRates (ins.cpp)
             buildRotationMatrix (ins.cpp)
             transformAccels (ins)
             transformWaterSpeed (ins)
********************************************************************/
 void fpeInsPosit(int sig)
{if (sig == SIGFPE) cerr << "floating point error in insPosit\n";}

Boolean
INS::insPosition(stampedSample& newSample)
{
   signal (SIGFPE, fpeInsPosit);

   double thetaA, phiA, xIncline, yIncline; // Working variables
   double waterSpeedCorrection[3]; // Filter correction for drift
                                   // and water speed
   if (sam1.getSample(newSample)) {

     applyBiasCorrections(newSample.sample);

     //Set output precision and fixed format
     cout.precision(6);
     cout.setf(ios::fixed);

     //Display linear accelrations and angular rates
     for (int j = 0; j < 8; j++) {
        gotoxy(59,j+1);
        cout << newSample.sample[j];
     }

     //Display time delta to the screen.
     gotoxy(9,25);
     cout << newSample.deltaT;

     xIncline = newSample.sample[0] / GRAVITY;
     yIncline = newSample.sample[1] / (GRAVITY * cos(posture[4]));
```

80

```cpp
if (fabs(yIncline) > 1.0) {
    static int inclineCount(0);
    gotoxy(1,24);
    cout << "Inclination errors: 0" << ++inclineCount << endl;
    return FALSE;
}

//Calculate low freq pitch and roll
thetaA = asin(xIncline);
phiA = -asin(yIncline);

//Transform body rates to euler rates.
transformBodyRates(newSample.sample);

//Calculate estimated roll rate (phi-dot).
velocities[3] = newSample.sample[3] + Kone1 * (phiA - posture[3]);
//Calculate estimated pitch rate (theta-dot).
velocities[4] = newSample.sample[4] + Kone2 * (thetaA - posture[4]);
//Calculate estimated heading rate (psi-dot).
velocities[5] = newSample.sample[5] + Ktwo * (newSample.sample[7] -
                posture[5]);

//integrate estimated pitch rate to obtain pitch angle
posture[3] += newSample.deltaT * velocities[3];
//integrate estimated roll rate to obtain roll angle
posture[4] += newSample.deltaT * velocities[4];
//integrate estimated yaw rate to obtain heading
posture[5] += newSample.deltaT * velocities[5];

//Display roll and pitch
gotoxy(8,17);
cout << (posture[3] * radToDeg);
gotoxy(8,18);
cout << (posture[4] * radToDeg);

buildRotationMatrix();

//Transform accels to earth coordinates
transformAccels(newSample.sample);

//Transform water speed to earth coordinates
transformWaterSpeed(newSample.sample[6], waterSpeedCorrection);

// Subtract out previous velocity and apply statistical gain
waterSpeedCorrection[0] = Kthree1 * (waterSpeedCorrection[0] -
                          velocities[0]);
waterSpeedCorrection[1] = Kthree2 * (waterSpeedCorrection[1] -
                          velocities[1]);

// Determine filtered accelerations
newSample.sample[0] += waterSpeedCorrection[0];
newSample.sample[1] += waterSpeedCorrection[1];
```

```
    //Integrate accelerations to obtain velocities
    velocities[0] += newSample.sample[0] * newSample.deltaT;
    velocities[1] += newSample.sample[1] * newSample.deltaT;
    velocities[2] += newSample.sample[2] * newSample.deltaT;

    //Integrate velocities to obtain posture
    posture[0] += (velocities[0] + current[0]) * newSample.deltaT;
    posture[1] += (velocities[1] + current[1]) * newSample.deltaT;
    posture[2] += velocities[2] * newSample.deltaT;

    newSample.current = sqrt(current[0] * current[0] +
                             current[1] * current[1]);

    newSample.sample[0] = posture[0];
    newSample.sample[1] = posture[1];
    newSample.sample[2] = posture[2];
    newSample.sample[3] = posture[3];
    newSample.sample[4] = posture[4];
    newSample.sample[5] = posture[5];

    //Display current location and posture
    for (j = 0; j < 6; j++) {
        gotoxy(52,j+12);
        cout << posture[j];
    }
    newSample.est.x = posture[0];
    newSample.est.y = posture[1];
    newSample.est.z = posture[2];

    return TRUE;
  }
  else {
    return FALSE;// New IMU information was unavailable.
  }
}
/********************************************************************
PROGRAM:correctPosition
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION: Reinitializes the INS based on a known position and compute
   apparent current based on past accumulated errors of the INS. It is
   called by the navigator each time a new GPS (true) fix is obtained.
RETURNS:void
CALLED BY: navPosit (nav)
CALLS:none
********************************************************************/

void
INS::correctPosition(const grid& truePosit, double positTime)
{
    double deltaT;

    // Correct for new day if necessary
    if (positTime < lastGPStime) {
```

```cpp
        positTime += 86400;
    }

    // Find time since last gps fix.
    deltaT = positTime - lastGPStime;

    // Detemine INS error since last gps fix
    double deltaX = truePosit.x - posture[0];
    double deltaY = truePosit.y - posture[1];

    // Reinitialize posture to known position (gps fix)
    posture[0] = truePosit.x;
    posture[1] = truePosit.y;
    posture[2] = 0.0; //Unit is assumed to be on the surface

    // Add gain filtered error to previous errors
    current[0] += Kfour1 * (deltaX / deltaT);
    current[1] += Kfour2 * (deltaY / deltaT);

    // Display new error current values
    for(int j = 0; j < 3; j++) {
        gotoxy(60,j+21);
        cout << current[j];
    }

    // Display updated posture
    for (j = 0; j < 6; j++) {
        gotoxy(52,j+12);
        cout << posture[j];
    }

    // Save the time of the gps fix for next calculation
    lastGPStime = positTime;
}
```

```
/*********************************************************************
PROGRAM:insSetUp
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Initializes the INS system. Sets the posture to the origin.
    Initializes the heading using magnetic compass information. Initilizes
    the times of the last GPS fix and last IMU information.
RETURNS:void
CALLED BY:initializeNavigator (nav)
CALLS:calulateBiasCorrections (ins)
      getSample (sampler)
      buildRotationMatrix (ins)
      transformWaterSpeed (ins)

*********************************************************************/
 void fpeInsSetUp(int sig)
{if (sig == SIGFPE) cerr << "floating point error in inSetUp\n";}

void
INS::insSetUp(double originTime, stampedSample* biasPackets)
{
    signal (SIGFPE, fpeInsSetUp);

    //Initialize the sampler
    sam1.initSampler();

    //set imu biases
    calculateBiasCorrections(biasPackets);

    //set initial true heading
    posture[5] = biasPackets[biasNumber-1].sample[7];

    //set initial speed
    buildRotationMatrix();
    transformWaterSpeed(biasPackets[biasNumber-1].sample[6], velocities);

    // initialize times
    lastGPStime = originTime;

    //Display initial error current values
    for(int j = 0; j < 3; j++) {
        gotoxy(60,j+21);
        cout << current[j];
    }
}
```

84

```
/*********************************************************************
PROGRAM:transformAccels
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Transforms linear accelerations from body coordinates to
   earth coordinates and removes the gravity component in the z direction.
RETURNS:void
CALLED BY:   navPosit
CALLS:       none
*********************************************************************/

void
INS::transformAccels (double newSample[])
{
   vector earthAccels;

   newSample[0] -= GRAVITY * sin(posture[4]);
   newSample[1] += GRAVITY * sin(posture[3]) * cos(posture[4]);
   newSample[2] += GRAVITY * cos(posture[3]) * cos(posture[4]);

   earthAccels = rotationMatrix * newSample;

   newSample[0] = earthAccels.element[0];
   newSample[1] = earthAccels.element[1];
   newSample[2] = earthAccels.element[2];

}

/*********************************************************************
PROGRAM:transformWaterSpeed
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Transforms water speed into a vector in earth coordinates
   and returns them in the speedCorrection variable.
RETURNS:void
CALLED BY:   navPosit
CALLS:       none
*********************************************************************/

void
INS::transformWaterSpeed (double waterSpeed, double speedCorrection[])
{
   double water[3] = {waterSpeed, 0.0, 0.0};
   vector waterVelocities = rotationMatrix * water;

   speedCorrection [0] = waterVelocities.element[0];
   speedCorrection [1] = waterVelocities.element[1];
   speedCorrection [2] = waterVelocities.element[2];
}
```

85

```
/*****************************************************************
PROGRAM: transformBodyRates
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:   Tranforms body euler rates to earth euler rates
RETURNS:    none
CALLED BY:  insPosit
CALLS:    buildBodyRateMatrix
*****************  ****************************************************/

void
INS::transformBodyRates (double newSample[])
{
    vector earthRates = buildBodyRateMatrix() * &(newSample[3]);

    newSample[3] = earthRates.element[0];
    newSample[4] = earthRates.element[1];
    newSample[5] = earthRates.element[2];
}

/*****************************************************************
PROGRAM: buildBodyRateMatrix
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:  Builds body to Euler rate translation matrix.
RETURNS:   rate translation matrix
CALLED BY: insPosit
CALLS:     none
*****************************************************************/

matrix
INS::buildBodyRateMatrix()
{
    matrix rateTrans;

    float tth = tan(posture[4]),
         sphi = sin(posture[3]),
         cphi = cos(posture[3]),
          cth = cos(posture[4]);


    rateTrans.element[0][0] = 1.0;
    rateTrans.element[0][1] = tth * sphi;
    rateTrans.element[0][2] = tth * cphi;
    rateTrans.element[1][0] = 0.0;
    rateTrans.element[1][1] = cphi;
    rateTrans.element[1][2] = -sphi;
    rateTrans.element[2][0] = 0.0;
    rateTrans.element[2][1] = sphi / cth;
    rateTrans.element[2][2] = cphi / cth;

    return rateTrans;
}
```

```
/*************************************************************************
PROGRAM: buildRotationMatrix
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Sets the body to earth coordinate rotation matrix.
RETURNS:void
CALLED BY:    insPosit
              insSetUp
CALLS:none
**************************************************************************/

void
INS::buildRotationMatrix()
{
    float spsi = sin(posture[5]),
          cpsi = cos(posture[5]),
          sth  = sin(posture[4]),
          sphi = sin(posture[3]),
          cphi = cos(posture[3]),
          cth  = cos(posture[4]);

    rotationMatrix.element[0][0] = cpsi * cth;
    rotationMatrix.element[0][1] = (cpsi * sth * sphi) - (spsi * cphi);
    rotationMatrix.element[0][2] = (cpsi * sth * cphi) + (spsi * sphi);
    rotationMatrix.element[1][0] = spsi * cth;
    rotationMatrix.element[1][1] = (cpsi * cphi) + (spsi * sth * sphi);
    rotationMatrix.element[1][2] = (spsi * sth * cphi) - (cpsi * sphi);
    rotationMatrix.element[2][0] = -sth;
    rotationMatrix.element[2][1] = cth * sphi;
    rotationMatrix.element[2][2] = cth * cphi;
}

/*************************************************************************
PROGRAM:post multiplication operator *
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Post multiply a 3 X 3 matrix times a 3 X 1 vector and
   return the result.
RETURNS:       3 X 1 vector
CALLED BY:
CALLS:None
**************************************************************************/

vector
operator* (matrix& transform, double state[])
{
    vector result;

    for (int i = 0; i < 3; i++) {

        result.element[i] = 0.0;

        for (int j = 0; j < 3; j++) {
```

87

```
            result.element[i] += transform.element[i][j] * state[j];
        }
    }
    return result;
}


/************************************************************************
PROGRAM:calculateBiasCorrections
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:  Calculates the initial imu bias by averaging a number of
           imu readings.
RETURNS:    none
CALLED BY:  insSetup
CALLS:none
************************************************************************/
void fpeCalculateBiasCorrections(int sig)
{if (sig == SIGFPE) cerr << "floating point error in
CalculateBiasCorrections\n";}

void
INS::calculateBiasCorrections(stampedSample* biasSample)
{
    signal (SIGFPE, fpeCalculateBiasCorrections);

    biasCorrection[3] = 0.0;
    biasCorrection[4] = 0.0;
    biasCorrection[5] = 0.0;

    for (int i = 0; i < biasNumber; i++) {

        while(!sam1.getSample(biasSample[i])) {/* */};
        biasCorrection[3] += biasSample[i].sample[3];
        biasCorrection[4] += biasSample[i].sample[4];
        biasCorrection[5] += biasSample[i].sample[5];
    }

    // Find the average of the biasNumber packets
    biasCorrection[3] = -(biasCorrection[3]/biasNumber);
    biasCorrection[4] = -(biasCorrection[4]/biasNumber);
    biasCorrection[5] = -(biasCorrection[5]/biasNumber);

    //Output the initial bias values
    for(int j = 3; j < 6; j++) {
        gotoxy(45,j+18);
        cout << biasCorrection[j];
    }
}
```

```
/*********************************************************************
PROGRAM: applyBiasCorrections
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:    Applies updated bias corrections to a sample.
RETURNS:     void
CALLED BY:   insPosit
CALLS:       none
*********************************************************************/

void
INS::applyBiasCorrections(double sample[])
{
    static const float biasWght(0.9999), sampleWght(0.0001);

    //Calculate updated bias values
    biasCorrection[3] = (biasWght * biasCorrection[3])
                        - (sampleWght * sample[3]);
    biasCorrection[4] = (biasWght * biasCorrection[4])
                        - (sampleWght * sample[4]);
    biasCorrection[5] = (biasWght * biasCorrection[5])
                        - (sampleWght * sample[5]);

    //Apply the bias to the sample
    sample[3] += biasCorrection[3];
    sample[4] += biasCorrection[4];
    sample[5] += biasCorrection[5];

    // Output the biases
    for(int j = 3; j < 6; j++) {
        gotoxy(45,j+18);
        cout << biasCorrection[j];
    }

}
```

J. SAMPLER .H

```
#ifndef __SAMPLER_H
#define __SAMPLER_H

#include <time.h>
#include <math.h>
#include <dos.h>
#include <conio.h>
#include <stdio.h>


#include "toetypes.h"
#include "location.h"
#include "a2d.h"
#include "compass.h"

#define MAX_SAMPLE_NUM 1000
const int INBUFFSIZE = 512;

/*********************************************************************
CLASS:sampler
AUTHOR:Eric Bachmann, Randy Walker
DATE:15 May 1996
FUNCTION:Formats, timestamps, low pass filters and limit checks IMU,
   water-speed and heading information.
COMMENTS: This class is extremely dependent upon the specific
   hardware configuration. It is designed to isolate to the INS from
   these particulars.
*********************************************************************/

class sampler {

   public:

      //Class Constructor
      sampler(): sampleIndex(0), subSampleIndex(0),
      samplePeriod(a2d1.chcnt * a2d1.delta_t * 0.000001){}

      //Initializes Sampler
      Boolean initSampler();

      //checks for the arrival of a new sample and formats it.
      Boolean getSample(stampedSample&);

   private:

      compass comp1;    //Compass instance

      a2dClass a2d1;    //A2D instance

      float sample[MAX_SAMPLE_NUM][8];

      int subSampleIndex;
```

```
        int sampleIndex;

        int sampleCount;

        float samplePeriod;

        Boolean readSamples(stampedSample& newSample);

        void filterSample(stampedSample& newSample);

        void formatSample(stampedSample& newSample);

        void increment(int& index)
            { if (++index == MAX_SAMPLE_NUM) index = 0;}

        void decrement(int& index)
            { if (--index < 0) index = MAX_SAMPLE_NUM - 1;}

};
#endif
```

K. SAMPLER.CPP

```cpp
#include <iostream.h>
#include <conio.h>
#include "sampler.h"

/*********************************************************************
PROGRAM:initSampler
AUTHOR:Eric Bachmann, Randy Walker
DATE:12 May 1995
MODIFIED: 15 May 1996 by Eric Bachmann & Randy Walker
FUNCTION:Initializes the compass and the A2D module.
RETURNS:TRUE
CALLED BY:
CALLS:          initCompass(), A2D member functions

*********************************************************************/
Boolean
sampler::initSampler()
{
    comp1.initCompass();

    a2d1.setRmsOff();
    a2d1.setSequencer();
    a2d1.lockTrigger();
    a2d1.resetFifo();
    a2d1.setFifo();
    a2d1.unlockTrigger();
    a2d1.setTrigger();

    return TRUE;
}
```

```
/*******************************************************************
PROGRAM: getSample
AUTHOR:Eric Bachmann, Randy Walker
DATE:15 May 1996
FUNCTION:Prepares raw sample data for use by the INS  object
RETURNS:TRUE, if a valid sample was obtained
CALLED BY:insPosit (ins)
          insSetup (ins)
CALLS:    Get (packetBuffer)
          createSample (sampler)
          formatSample (sampler)
          inLimitSample (sampler)
*******************************************************************/

//checks for the arrival of a new sample
Boolean
sampler::getSample(stampedSample& newSample)
{

    if (readSamples(newSample)) {

        filterSample(newSample);

        formatSample(newSample);

        return TRUE;

    }

    return FALSE;   //Sample packet not available
}
```

```
/**********************************************************************
PROGRAM: readeSamples
AUTHOR:Eric Bachmann, Randy Walker
DATE:12 May 1996
FUNCTION: Retrieves all samples of the IMU, water speed, and depth
    that are present in the A2D FIFO until the FIFO is EMPTY.  Calculates
    delta_t.
RETURNS:  Boolen: TRUE - There were new samples pulled from the FIFO
                  FALSE - There were no new samples
CALLED BY: getSample
CALLS:      getFifoStatus(), getFifoData()
**********************************************************************/
Boolean
sampler::readSamples(stampedSample& newSample)
{

    //Did the FIFO overflow?
    if (a2d1.getFifoStatus() == FULL) {
       gotoxy(17,20);
       cout << "FIFO Overflowed, execution terminated..." << endl;
       exit(1);
    }

    //Does the FIFO have new samples?
    if (a2d1.getFifoStatus() != EMPTY) {

       sampleCount = 0;                //Counts the number of samples taken

       //Empty the FIFO
       while (a2d1.getFifoStatus() != EMPTY) {


          sample[sampleIndex][subSampleIndex++] = a2d1.getFifoData();

          //Has it pulled one sample of each channel from the FIFO?
          if (subSampleIndex == 8) {
             subSampleIndex= 0;
             increment(sampleIndex); //set to record next sample
             ++sampleCount;
          }

       }

       if (sampleCount > 0) {

          //calculate time delta
          newSample.deltaT = sampleCount * samplePeriod;

          gotoxy(1,19);
          printf("        ");
          gotoxy(1,19);
          printf("sampleCount: %d",sampleCount);
          return TRUE;
       }
```

```
        else {
            return FALSE;
        }
    }

    else {

        //No new samples
        return FALSE;
    }
}

/**********************************************************************
PROGRAM: createSample
AUTHOR:Eric Bachmann, Randy Walker
DATE:15 May 1996
FUNCTION:  Low pass filters by averaging over all samples received since
           the last sample..
RETURNS:   void
CALLED BY: getSample
CALLS:     none
**********************************************************************/
void
sampler::filterSample(stampedSample& newSample)
{

    for (int i = 0; i < 8; i++) {
        newSample.sample[i] = 0;
    }

    int j(sampleIndex);

    for (i = 0; i < sampleCount; i++) {

        decrement(j);
        newSample.sample[0] += sample[j][0] / sampleCount;
        newSample.sample[1] += sample[j][1] / sampleCount;
        newSample.sample[2] += sample[j][2] / sampleCount;
        newSample.sample[3] += sample[j][3] / sampleCount;
        newSample.sample[4] += sample[j][4] / sampleCount;
        newSample.sample[5] += sample[j][5] / sampleCount;
        newSample.sample[6] += sample[j][6] / sampleCount;
        newSample.sample[7] += sample[j][7] / sampleCount;
    }

}
```

```
/**********************************************************************
PROGRAM:formatSample
AUTHOR:Eric Bachmann, Randy Walker
DATE:15 May 1996
FUNCTION:Converts integers representing voltage readings into
    units which are useable by the INS.
RETURNS: void
CALLED BY:getSample
CALLS:none
**********************************************************************/

//Calls the methods to convert the voltages to real world units
void
sampler::formatSample (stampedSample& newSample)
{
    newSample.sample[0] = xAccelUnits(newSample.sample[0]);
    newSample.sample[1] = yAccelUnits(newSample.sample[1]);
    newSample.sample[2] = zAccelUnits(newSample.sample[2]);

    newSample.sample[3] = pUnits(newSample.sample[3]);
    newSample.sample[4] = qUnits(newSample.sample[4]);
    newSample.sample[5] = rUnits(newSample.sample[5]);

    newSample.sample[6] = waterSpeedUnits(newSample.sample[6]);
    newSample.sample[7] = comp1.getHeading();
}
```

L. COMPASS.H

```
#ifndef _COMPASS_H
#define _COMPASS_H

#include <iostream.h>
#include <fstream.h>
#include <conio.h>

#include "portbank.h"
#include "toetypes.h"
#include "location.h"
#include "compBuff.h"

/*********************************************************************
   CLASS:compass
   AUTHOR:Eric Bachmann, Dave Gay
   DATE:11 July 1995
   MODIFIED: 15 May, 1996 by Eric Bachmann & Randy Walker
   FUNCTION:Reads compass messages from the compass buffer. Checks for
   validchecksum. Corrects heading for megnetic variation. Heading is
   continous. There is no branch cut at 360 degrees.
*********************************************************************/

class compass {

  public:

    //Class Constructor
    compass() : port2(COMports.Init(COM2, BYTE(3), b9600,
             NOPARITY, BYTE(8), BYTE(1), NONE, headings))
             {}

    //Initialize currentHeading
    float initCompass();

    //returns the latest heading
    float getHeading();

  private:

    //buffer for compass data
    compBuffer headings;

    //Maintains the most recently obtained heading.
    float currentHeading;

    //instantiates serial port communications on comm2
    bufferedSerialPort& port2;

    //calculates the check sum of the message
    Boolean checkSumCheck(const compData&);

    //Parses the heading out of a compass message.
```

97

```
    float parseCompData(const compData&, const BYTE);

    // Convert magnetic direction based on magnetic variation.
    float trueHeading(const float);

    // Returns the heading without branch cuts
    float continousHeading(const float);

};

#endif
```

M. COMPASS.CPP

```cpp
#include <math.h>
#include "compass.h"

float
compass::initCompass()
{
    float tempHeading;
    compData rawMessage;

    while ((headings.Get(rawMessage)==FALSE)
            || (checkSumCheck(rawMessage)==FALSE)) {}

    tempHeading = parseCompData(rawMessage, 'C') * degToRad;
    currentHeading = continousHeading(trueHeading(tempHeading));

    return currentHeading;
}
```

```
/*********************************************************************
   NAME:getHeading
   AUTHOR:  Eric Bachmann, Dave Gay
   DATE:11 July 1995
   FUNCTION:
   Determines if an updated gps position message is available and
   copies it into the input argument 'rawMessage'. If the message
   has a valid checksum and was obtained with atleast three
   satelites in view,  a 'TRUE' is returned to the caller,
   indicating that the message is valid.
   RETURNS:     TRUE, if a valid position message is contained in the
            input argument.
   CALLED BY:   navPosit (navigator.h)
   CALLS:Get (buffer.h)
         checkSumCheck (gps.h) .
*********************************************************************/

float
compass::getHeading()
{
    float tempHeading;
    Boolean checkSumFlag;
    compData rawMessage;

    if ((headings.Get(rawMessage)) && (checkSumCheck(rawMessage))) {

      tempHeading = parseCompData(rawMessage, 'C') * degToRad;
      currentHeading =
          continousHeading(trueHeading(tempHeading));

      return currentHeading;
    }
    else {
      return currentHeading; // No updated position is available.
    }

}

BYTE
asciiToHex(BYTE letter)
{

    if (letter >= 'A') {

      return (letter - 'A' + 10);
    }
    else {

      return (letter - 48);
    }
}
```

```
/***********************************************************************
    PROGRAM:checkSumCheck
    AUTHOR:Eric Bachmann, Dave Gay
    DATE:11 July 1995
    FUNCTION:
    Takes an exclusive or of bytes 2 through 78 in a Motorola format (@@EA)
    position message and compares it to the checksum of the message of the
    message.
    RETURNS: TRUE, if the message contains a valid checksum
    CALLED BY:   gpsPosition (gps)
    CALLS:       none
***********************************************************************/

Boolean
compass::checkSumCheck(const compData& newMessage)
{
    BYTE calChkSum(0);
    BYTE mesChkSum(0);

    for (int i = 1; newMessage[i] != '*'; i++) {

            calChkSum ^= newMessage[i];
    }

    mesChkSum = asciiToHex(newMessage[i+1]) * 16
                + asciiToHex(newMessage[i+2]);


    return Boolean(calChkSum == mesChkSum);
}
```

100

```
/**********************************************************************
   PROGRAM:    trueHeading
   AUTHOR:     Eric Bachmann, Dave Gay
   DATE:       11 July 1995
   FUNCTION:   Convert magnetic direction to true based on local magnetic
         variation.
   RETURNS:    true heading
   CALLED BY:  insPosit
         insSetUp
   CALLS:      none
**********************************************************************/

float
compass::trueHeading(const float magHeading)
{
   static double twoPi(2.0 * M_PI);
   double trueHeading = magHeading + RADIANMAGVAR;

   if (trueHeading > twoPi) {
     trueHeading -= twoPi;
   }

   return trueHeading;

}
```

```
/********************************************************************
    PROGRAM:     continousHeading
    AUTHOR:      Eric Bachmann
    DATE:        11 July 1995
    FUNCTION:    Maintains track of branch cuts and returns a continous heading.
    RETURNS:     continous true heading
    CALLED BY:   insPosit
                 insSetUp
    CALLS:       none
********************************************************************/

float
compass::continousHeading(const float trueHeading)
{
    const float twoPi(2.0 * M_PI);
    static int branchCutCount(0);
    static float previousHeading(trueHeading);

    if ((4.71 < previousHeading) && (trueHeading < 1.57)){
       ++branchCutCount; //Went through North in a right hand turn
    }
    else {
      if ((1.57 > previousHeading) && (trueHeading > 4.71)) {
         --branchCutCount;//Went through North in a left hand turn
      }
    }

    previousHeading = trueHeading;

    return trueHeading + (branchCutCount * twoPi);
}
```

102

```
/**********************************************************************
    PROGRAM:    parseHeading
    AUTHOR:     Eric Bachmann
    DATE:       11 July 1995
    FUNCTION:   Parses the heading out of a compass message.
    RETURNS:    the message heading as a float
    CALLED BY:  insPosit
                insSetUp
    CALLS:      none
**********************************************************************/

float
compass::parseCompData(const compData& rawMessage, const BYTE key)
{
    float dataSum(0);

    for(int j = 0; rawMessage[j] != key; j++){}

    j++;

    for(int i = 0; rawMessage[i + j] != '.'; i++){}

    switch (i) {

        case 3:

            dataSum = (rawMessage[j] - 48) * 100.0 +
                      (rawMessage[j+1] - 48) * 10.0 +
                      (rawMessage[j+2] - 48) +
                      (rawMessage[j+4] - 48) * 0.1;
            break;

        case 2:

            dataSum = (rawMessage[j] - 48) * 10.0 +
                      (rawMessage[j+1] - 48) +
                      (rawMessage[j+3] - 48) * 0.1;
            break;

        case 1:

            dataSum = (rawMessage[j] - 48) +
                      (rawMessage[j+2] - 48) * 0.1;

            break;
    }

    return dataSum;

}
```

N. A2D.H

```c
#ifndef __A2D_H
#define __A2D_H

#include <dos.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <iostream.h>
#include <fstream.h>


//ESP A2D General Global Definitions
#define DEFBASE   0x100   //Base address SEL=1->0x300 & SEL=0->0x100
#define FIFOSIZE  1000   //FIFO size (MAX=1000 decimal)
#define  MAXCHAN  0x10   //Max channels

//ESP A2D Status Register Definitions
//BASE+02h: 011D DDDD
#define INT_STAT 0x10    // 0001 0000  INTERRUPT STATUS   (1=IRQ Pending)
#define TRG_STAT 0x08    // 0000 1000  TRIGGER STATUS     (1=Triggered)

#define FULL       0x01      // 0000 0001 FIFO FULL        (001=Full)
#define HALF       0x05      // 0000 0101 FIFO HALF FULL   (101=Half Full)
#define EMPTY      0x06      // 0000 0110 FIFO EMPTY       (110=Empty)

//ESP A2D Control Register Definitions
//BASE+08h: DDDD DDDD
//BASE+09h: DDDD DDRR
#define GATE1OUT 0x0008 // 0000 0000 0000 1000 GATE1OUT   (Always Driven)

#define TRG_POS  0x0010 // 0000 0000 0001 0000 TRIG POS   (Trig on +|-)
#define SET_TRG  0x0020 // 0000 0000 0010 0000 TRIG SET   (Active  LOW)
#define RST_TRG  0x0040 // 0000 0000 0100 0000 TRIG CLR   (Active  LOW)
#define INT_EN   0x0080 // 0000 0000 1000 0000 IRQ ENAB   (Active HIGH)

#define DIFF        0x0400  // 0000 0100 0000 0000 DIFF/SE   (1=DIFF 0=SE)
#define RMS         0x0800  // 0000 1000 0000 0000 RMS Mode  (1=ON   0=OFF)

#define CAL         0x1000  // 0001 0000 0000 0000 CAL  Mode (1=ON   0=OFF)
#define PRG_SEQ     0x1000  // 0001 0000 0000 0000 SEQ  Mode (1=PRG  0=RUN)
#define ACDC        0x2000  // 0010 0000 0000 0000 ACDC Mode (1=DC   0=AC)
#define SAM_SEQ     0x4000  // 0100 0000 0000 0000 SAMP/SEQ  (1=SEQ  0=SAMP)
#define RST_FIFO    0x8000  // 1000 0000 0000 0000 FIFO Reset(1=EN   0=REW)


//ESP A2D Useful Definitions
#define CLRRATE   0xFFF8 // CLEAR RATE TO HIGHEST RATE
```

```
//*********************************************************************
// CLASS NAME: a2dClass
// AUTHOR: Randy Walker
// DATE: 27 March 1996
// DESCRIPTION: Provides for the software operation of the A2D module.
//*********************************************************************

//Class Definition for the A2D Class
class a2dClass {

    public:

        //Class Constructor; reads a2d.cfg file, initializes hardware
        a2dClass();

        //Reads a2d.cfg configuration file
        void readConfigFile();

        //Sets address mapping
        void initSysAddr(void);

        //Initializes the A2D Control Register
        void initHardware(void);

        //Print out the variable ctrlw, for debug purposes
        void printCtrlw(void);

        //Sets the A2D Control Register for Single-Ended mode
        void setSe(void);

        //Sets the A2D Control Register for Differential mode
        void setDiff(void);

        //Loads sequencer memory with channel data
        void setChannel(unsigned seq,unsigned ch,unsigned g10,unsigned g2);

        //Sets sequencer to program mode
        void setProgSeq(void);

        //Sets sequencer to run mode
        void setRunSeq(void);

        //Loads sequencer address counter with number of channels to scan.
        void setCount(unsigned nch);

        //Sets AC or DC Coupling
        void setAcDc(unsigned acdc);

        //Prevents triggering
        void lockTrigger(void);

        //Allow the triger to function
        void unlockTrigger(void);
```

```c
//Toggle the trigger (software triggering)
void setTrigger(void);

//Clears the trigger
void resetTrigger(void);


//Switches in the RMS measurement chip
void setRmsOn(void);

//Switches out RMS measurement chip
void setRmsOff(void);

//Sets the A2D module to sequencer mode
void setSequencer(void);

//Sets the A2D module to sampler mode
void setSamplerRate(unsigned);

//Set GATE1OUT bit of control word high
void gate1outOn(void);

//Set GATE1OUT bit of control word low
void gate1outOff(void);

//Sets timer channel 1 to square-wave input
void squareWaveTimer1(unsigned);

//Initialize the A2D timing using timer 2
void initTiming(unsigned dt);

//Rewind FIFO to beginning of memory
void resetFifo(void);

//Enable FIFO to acquire data
void setFifo(void);

//Returns th state of the fifo
unsigned getFifoStatus(void);

//Returns next data word stored in FIFO
signed getFifoData(void);

//Program timer channel 0 to set the desired interrupt rate
void setIntRate(unsigned intrate);

//Locksout the interupt request line
void intOff(void);

//Enables system interuppt request
void intOn(void);

//Sets the trigger level; trigger level (0=-10V, 128=0V, 255=+10V)
void setTriggerLevel(unsigned tl);
```

```cpp
        //Sets falling or rising edge trigger
        void setTriggerPosition(unsigned tp);

        //Calibrates zero offset error
        void zeroOffset(void);

        //Grounds the two differential inputs for zero adjust
        void grndInput(void);

        //Ungrounds the two differential inputs
        void freeInput(void);

        //Adjust the trimmer on the PGA
        void zeroAdjust(void);

        int chcnt;                      //Number of channels to sequence
        unsigned delta_t;               //period between channels

    private:

        unsigned ctrlw;                 //Holds A2D Control Register update values
        unsigned seqcnt;                //Sequence Counter
        unsigned mode_sel;              //Single-ended or Differential
        unsigned mode_acdc;             //AC/DC Coupling
        unsigned samprate;              //Sample Rate in Recurrent Mode
        unsigned sampindex;             //Which Channel to Sample in Recurrent Mode
        unsigned seqaddr[MAXCHAN];      //Sequencer Address
        unsigned chan[MAXCHAN];         //Channel
        unsigned g10[MAXCHAN];          //x10 Gain
        unsigned g2[MAXCHAN];           //x2 Gain

};

#endif
```

O. A2D.CPP

```cpp
#include "a2d.h"

//ESP A2D Addresses
unsigned BASE    = DEFBASE;     // BASE I/O  ADDR          [BASE]   ()
unsigned FIFO    = 0x00;        // FIFO READ ADDR          [00-01]  (R)
unsigned MEM     = 0x00;        // SEQUENCER ADDR          [00-01]  (W)
unsigned STAT    = 0x02;        // STATUS REGISTER         [02]     (R)
unsigned COUNT   = 0x02;        // SEQUENCER ADDR PTR      [02]     (W)
unsigned TIMER0  = 0x04;        // TIMER 0                 [04]     (R/W)
unsigned TIMER1  = 0x05;        // TIMER 1                 [05]     (R/W)
unsigned TIMER2  = 0x06;        // TIMER 2                 [06]     (R/W)
unsigned TIMERC  = 0x07;        // TIMER CONTROL WORD      [07]     (R/W)
unsigned CNTL    = 0x08;        // A2D CONTROL REGISTER    [08-09]  (W)
unsigned DAC     = 0x0C;        // DAC DATA                [0C]     (W)
```

```cpp
//*********************************************************************
// FUNCTION NAME: a2dClass()
// AUTHOR: Randy Walker
// DATE: 27 March 1996
// DESCRIPTION: Sets defaults, reads a2d.cfg file, initializes address map
//              and hardware
// RETURNS: void
// CALLS: readConfigFile(), initSysAddr(), initHardware()
// CALLED BY: Object declaration
//*********************************************************************
a2dClass::a2dClass(void)
{
   ctrlw=0;
   seqcnt=1;
   mode_sel=0;
   mode_acdc=1;
   delta_t=3;
   chcnt=1;
   samprate=0;
   sampindex=0;
   readConfigFile();
   initSysAddr();
   initHardware();

}


//*********************************************************************
// FUNCTION NAME: readConfigFile()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Reads the a2d.cfg file and sets variables
// RETURNS: void
// CALLS: none
// CALLED BY: a2d class constructor
//*********************************************************************
void a2dClass::readConfigFile()
{
   FILE *configFile;
   char junk[128];

   if ((configFile = fopen("a2d.cfg", "r")) == NULL){
      fprintf(stderr, "Cannot open file A2D.CFG...\n");
      exit(1);
   }

   fscanf(configFile,"%x%s",&seqcnt,junk);
   if (seqcnt==0 ||seqcnt>0x0F){  //seqcnt must be 1-F (15 max in seq mode)
      cout << "\nseqcnt out of range in A2D.CFG...\n";
      exit(1);
   }

   fscanf(configFile,"%d%s",&mode_sel,junk);
   if (mode_sel !=0 && mode_sel != 1){
      cout << "\nmode_sel out of range in A2D.CFG...\n";
```

108

```
    exit(1);
}

fscanf(configFile,"%d%s",&mode_acdc,junk);
if (mode_acdc !=0 && mode_acdc != 1){
    cout << "\nmode_acdc out of range in A2D.CFG...\n";
    exit(1);
}

fscanf(configFile,"%x%s",&chcnt,junk);
if (chcnt == 0 || chcnt > 0x0F){ //chcnt must be 1-F (15 max in seq mode)
    cout << "\nchcnt out of range in A2D.CFG...\n";
    exit(1);
}

fscanf(configFile,"%d%s",&delta_t,junk);
if (delta_t < 3 || delta_t > 8192){
    cout << "\ndelta_t out of range in A2D.CFG...\n";
    exit(1);
}

if (delta_t < 6 && chcnt > 1){
    cout << "\ndelta_t must be > 6 for chcnt > 1...\n";
    exit(1);
}

fscanf(configFile,"%d%s",&samprate,junk);
if (samprate > 7){
    cout << "\nsamprate out of range in A2D.CFG...\n";
    exit(1);
}

fscanf(configFile,"%x%s",&sampindex,junk);
if (sampindex > 0x0F){
    cout << "\nsampindex out of range in A2D.CFG...\n";
    exit(1);
}

for (int i=0;i<seqcnt;i++)
    fscanf(configFile,"%x%x%x%x",&seqaddr[i],&chan[i],&g10[i],&g2[i]);
fclose(configFile);

}
```

```
//********************************************************************
// FUNCTION NAME: initSysAddr()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets system address mappings
// RETURNS: void
// CALLS: none
// CALLED BY: a2d class constructor
//********************************************************************
void a2dClass::initSysAddr(void)
{
    //clear BASE
    FIFO    &= 0x0F;        // FIFO READ ADDRESS    [00,01] (R)
    MEM     &= 0x0F;        // SEQENCER MEM ADDRESS [00,01] (W)
    STAT    &= 0x0F;        // STATUS REGISTER      [02]    (R)
    COUNT   &= 0x0F;        // SEQENCER ADDRESS PTR [02]    (W)
    TIMER0  &= 0x0F;        // TIMER 0              [04]    (R/W)
    TIMER1  &= 0x0F;        // TIMER 1              [05]    (R/W)
    TIMER2  &= 0x0F;        // TIMER 2              [06]    (R/W)
    TIMERC  &= 0x0F;        // TIMER CONTROL WORD   [07]    (R/W)
    CNTL    &= 0x0F;        // CONTROL REGISTER     [08]    (R/W)
    DAC     &= 0x0F;        // DAC DATA             [0C]    (W)

    //set BASE
    FIFO    |= BASE;        // FIFO READ ADDRESS    [00,01] (R)
    MEM     |= BASE;        // SEQENCER MEM ADDRESS [00,01] (W)
    STAT    |= BASE;        // STATUS REGISTER      [02]    (R)
    COUNT   |= BASE;        // SEQENCER ADDRESS PTR [02]    (W)
    TIMER0  |= BASE;        // TIMER 0              [04]    (R/W)
    TIMER1  |= BASE;        // TIMER 1              [05]    (R/W)
    TIMER2  |= BASE;        // TIMER 2              [06]    (R/W)
    TIMERC  |= BASE;        // TIMER CONTROL WORD   [07]    (R/W)
    CNTL    |= BASE;        // CONTROL REGISTER     [08]    (R/W)
    DAC     |= BASE;        // DAC DATA             [0C]    (W)

}
```

```
//*********************************************************************
// FUNCTION NAME: initHardware()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets the A2D Control Register to 0020 and sets the
//              data member, ctrlw=0060; initializes the module setup
//              for software triggering of the A2D.  Programs each
//              channel.
// RETURNS: void
// CALLS: outpw()
// CALLED BY: a2d class constructor
//*********************************************************************
void a2dClass::initHardware(void)
{
   outpw(CNTL,SET_TRG);
   ctrlw = SET_TRG|RST_TRG;

   if(mode_sel == 0)
      setSe();

   else
      setDiff();

   for(int i = 0;i < chcnt;i++){
      setChannel(seqaddr[i],chan[i],g10[i],g2[i]);
   }
   setAcDc(mode_acdc);
   initTiming(delta_t);
   setCount(chcnt);

}

//*********************************************************************
// FUNCTION NAME: printCtrlw()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Print A2D control register var, ctrlw.
//              The variable is used to set a byte in the
//              ESP A2D control register at BASE + 08h/09h
//              Used during application code debug
// RETURNS: void
// CALLS: none
// CALLED BY: none
//*********************************************************************
void a2dClass::printCtrlw(void)
{
   printf("ctrlw: %04x\t", ctrlw);
   for (int i=0x00;i<0x10;i++){
      printf("%i",((ctrlw>>0x0F-i) & 1));
      if ((i+1)%4==0)
         printf(" ");
   }

}
```

111

```
//******************************************************************
// FUNCTION NAME: setSe()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets ctrlw for single ended mode and writes ctrlw to
//              A2D Control Register
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initHardware()
//******************************************************************
void a2dClass::setSe(void)
{
   ctrlw &= ~DIFF;
   outpw(CNTL,ctrlw);


}


//******************************************************************
// FUNCTION NAME: setDiff()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets ctrlw for differential mode and writes ctrlw to
//              A2D Control Register
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initHardware()
//******************************************************************
void a2dClass::setDiff(void)
{
   ctrlw |= DIFF;
   outpw(CNTL,ctrlw);


}
```

```
//*****************************************************************
// FUNCTION NAME: setChannel()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Loads sequencer memory with channel data
// CALLS: progSeq(), outpw(), runSeq()
// CALLED BY: initHardware()
// VARIABLES: seq - sequencer number
//            ch  - channel number
//            g10 - x10 gain value
//            g2  - x2 gain value
//*****************************************************************
void a2dClass::setChannel(unsigned seq,unsigned ch,unsigned g10,unsigned g2)
{
    unsigned d = 0;

    setProgSeq();                // set sequencer program mode
    outpw(COUNT,seq);            // set sequencer address

    //load sequencer memory
    d |= ch<<8;                  // channel
    d |= (g2<<12);               // gain X2
    d |= (g10<<14);              // gain X10
    outpw(MEM,d);                // load sequencer

    setRunSeq();                 // set sequencer run mode

}


//*****************************************************************
// FUNCTION NAME: setProgSeg()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets sequencer to program mode
// RETURNS: void
// CALLS: outpw()
// CALLED BY: setChannel()
//*****************************************************************
void a2dClass::setProgSeq(void)
{
    ctrlw |= PRG_SEQ;
    outpw(CNTL,ctrlw);

}
```

```
//********************************************************************
// FUNCTION NAME: setRunSeq()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets sequencer to run mode
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//********************************************************************
void a2dClass::setRunSeq(void)
{
   ctrlw &= ~PRG_SEQ;
   outpw(CNTL,ctrlw);


}


//********************************************************************
// FUNCTION NAME: setCount()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Loads sequencer address counter with number of channels
//              to scan.
// RETURNS: void
// CALLS: outpw(), setProgSeq(), setRunSeq()
// CALLED BY: initHardware()
// VARIABLES: nch - number of channels to sequence
//********************************************************************
void a2dClass::setCount(unsigned nch)
{
   nch=nch<<4;           // put in upper nibble
   outpw(COUNT,nch);     // out to register
   setProgSeq();         // reset sequencer
   setRunSeq();          // put it in run mode


}
//********************************************************************
// FUNCTION NAME: setAcDc()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets AC or DC Coupling
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initHardware()
// VARIABLES: acdc - holds coupling value
//********************************************************************
void a2dClass::setAcDc(unsigned acdc)
{
   if (acdc)
      ctrlw |= ACDC;        // acdc=1 -> DC
   else
      ctrlw &= ~ACDC;       // acdc=0 -> AC
   outpw(CNTL,ctrlw);


}
```

114

```
//*********************************************************************
// FUNCTION NAME: lockTrigger()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Prevents triggering
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initSampler()
//*********************************************************************
void a2dClass::lockTrigger(void)
{
    ctrlw &= ~RST_TRG;
    outpw(CNTL,ctrlw);

}


//*********************************************************************
// FUNCTION NAME: unlockTrigger()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Allow the trigger to function
// RETURNS: void
// CALLS: outpw()
// CALLED BY: readSmaple()
//*********************************************************************
void a2dClass::unlockTrigger(void)
{
    ctrlw |= RST_TRG|SET_TRG;
    outpw(CNTL,ctrlw);

}


//*********************************************************************
// FUNCTION NAME: setTrigger()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Toggle the trigger (software triggering)
// RETURNS: void
// CALLS: outpw()
// CALLED BY:readSample()
//*********************************************************************
void a2dClass::setTrigger(void)
{
    outpw(CNTL,ctrlw&~SET_TRG|RST_TRG);
    outpw(CNTL,ctrlw| SET_TRG|RST_TRG);

}
```

```
//*******************************************************************
// FUNCTION NAME: resetTrigger()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Clears the trigger
// RETURNS: void
// CALLS: outpw()
// CALLED BY: readSample()
//*******************************************************************
void a2dClass::resetTrigger(void)
{
    outpw(CNTL,ctrlw|SET_TRG&~RST_TRG);
    outpw(CNTL,ctrlw|SET_TRG| RST_TRG);

}


//*******************************************************************
// FUNCTION NAME: setRmsOn()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Switches in the RMS measurement chip
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*******************************************************************
void a2dClass::setRmsOn(void)
{
    ctrlw |= RMS;
    outpw(CNTL,ctrlw);

}


//*******************************************************************
// FUNCTION NAME: setRmsOff()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Switches out RMS measurement chip
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initSampler()
//*******************************************************************
void a2dClass::setRmsOff(void)
{
    ctrlw &= ~RMS;
    outpw(CNTL,ctrlw);

}
```

116

```cpp
//*******************************************************************
// FUNCTION NAME: setSequencer()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets the A2D module to sequencer mode
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*******************************************************************
void a2dClass::setSequencer(void)
{
   ctrlw |= SAM_SEQ;
   outpw(CNTL,ctrlw);

}


//*******************************************************************
// FUNCTION NAME: setSamplerRate()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets the A2D module to sampler mode
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
// VARIABLES: rate - sampler rate
//*******************************************************************
void a2dClass::setSamplerRate(unsigned rate)
{
   ctrlw &= ~SAM_SEQ;     //Set to sampler mode
   ctrlw &= CLRRATE;      //Clear previous rate to 000
   ctrlw |= rate;         //Set new rate
   outpw(CNTL,ctrlw);     //Set Control Word

}


//*******************************************************************
// FUNCTION NAME: gate1outOn()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Set GATE1OUT bit of control word high
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*******************************************************************
void a2dClass::gate1outOn(void)
{
   ctrlw |= GATE1OUT;
   outpw(CNTL,ctrlw);

}
```

```
//*****************************************************************
// FUNCTION NAME: gate1outOff()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Set GATE1OUT bit of control word low
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*****************************************************************
void a2dClass::gate1outOff(void)
{
   ctrlw &= ~GATE1OUT;
   outpw(CNTL,ctrlw);

}


//*****************************************************************
// FUNCTION NAME: squareWaveTimer1()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets timer channel 1 to square-wave input
// RETURNS: void
// CALLS: outp()
// CALLED BY: none
// VARIABLES: dt-micro seconds per period (1 to 8192)
//                   assuming 8 MHz clock input
//             ch-timer channel 1
//             ph-local variable
//             pl-local variable
//*****************************************************************
void a2dClass::squareWaveTimer1(unsigned dt)
{
   char    ph,pl;

   pl = (dt*8)&0xFF;         // 8 CLOCKS PER uS
   ph = (dt*8)>>8;

   outp(TIMERC,0x76);        // initialize timer
   outp(TIMER1,pl);          // dt uS delay
   outp(TIMER1,ph);          // with 8 MHz clock

}
```

118

```
//*******************************************************************
// FUNCTION NAME: initTiming()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Initialize the A2D timing using timer 2
// RETURNS: void
// CALLS: outp()
// CALLED BY: initHardware()
// VARIABLES: dt - number of micro seconds (3 to 2730)
//*******************************************************************
void a2dClass::initTiming(unsigned dt)
{
    char     ph,pl;

    pl = (dt*8)&0xFF;     // 8 CLOCKS PER uS
    ph = (dt*8)>>8;

    outp(TIMERC,0xB6);    // initialize timer2
    outp(TIMER2,pl);      // dt uS delay
    outp(TIMER2,ph);      // with 8 MHz clock

}


//*******************************************************************
// FUNCTION NAME: resetFifo()
// AUTHOR: Randy Walker, based on [MAXUS 95] code
// DATE: 27 March 1996
// DESCRIPTION: Rewind FIFO to beginning of memory
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*******************************************************************
void a2dClass::resetFifo(void)
{
    ctrlw &= ~RST_FIFO;
    outpw(CNTL,ctrlw);

}


//*******************************************************************
// FUNCTION NAME: setFifo()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Enable FIFO to acquire data
// RETURNS: void
// CALLS: outpw()
// CALLED BY: initSampler()
//*******************************************************************
void a2dClass::setFifo(void)
{
    ctrlw |= RST_FIFO;
    outpw(CNTL,ctrlw);

}
```

```
//*********************************************************************
// FUNCTION NAME: getFifoStatus()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Returns FIFO status
// RETURNS: RETURNS: 6 - empty
//                                      5 - half full
//                                      1 - full
// CALLS: inpw()
// CALLED BY: readSample()
//*********************************************************************
unsigned a2dClass::getFifoStatus(void)
{
   return (inpw(STAT)&7);

}


//*********************************************************************
// FUNCTION NAME: getFifoData()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Returns next data word stored in FIFO
// RETURNS: 16bits of data. Lower 12 are A2D data
// CALLS: inpw()
// CALLED BY: readSample()
//*********************************************************************
signed a2dClass::getFifoData(void)
{
   return (inpw(FIFO)&0x0FFF);       //Get data and mask upper nibble

}


//*********************************************************************
// FUNCTION NAME: setIntRate()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Program timer channel 0 to set the desired interrupt
//              rate
// RETURNS: void
// CALLS: outp()
// CALLED BY: none
// VARIABLES: intrate-micro secs per period (1 to 8192)
//            assuming 8 MHz clock input
//*********************************************************************
void a2dClass::setIntRate(unsigned intrate)
{
   outp(TIMERC,0x36);                // Set timer 0 to mode 3
   outp(TIMER0,(intrate*8)&0xFF);  // Load Least Significant Byte
   outp(TIMER0,(intrate*8)>>8);    // Load Most Significant Byte

}
```

```cpp
//*******************************************************************
// FUNCTION NAME: intOff()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Locksout the interupt request line
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*******************************************************************
void a2dClass::intOff(void)
{
   ctrlw &= ~INT_EN;          // INT_EN is active high
   outpw(CNTL,ctrlw);

}


//*******************************************************************
// FUNCTION NAME: intOn()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Enables system interuppt request
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*******************************************************************
void a2dClass::intOn(void)
{
   ctrlw |= INT_EN;           // INT_EN is active high
   outpw(CNTL,ctrlw);

}


//*******************************************************************
// FUNCTION NAME: setTriggerLevel()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets the trigger level
// RETURNS: void
// CALLS: outp()
// CALLED BY: none
// VARIABLES: tl-trigger level  (0=-10V, 128=0V, 255=+10V)
//*******************************************************************
void a2dClass::setTriggerLevel(unsigned tl)
{
   outp(DAC,tl);

}
```

```
//********************************************************************
// FUNCTION NAME: setTriggerPosition()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Sets falling or rising edge trigger
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
// VARIABLES: tp: 0=falling, 1=rising
//********************************************************************
void a2dClass::setTriggerPosition(unsigned tp)
{
   ctrlw &= ~TRG_POS;         //Clear previous TRG_POS
   ctrlw |= (tp)?TRG_POS:0; //Evaluate tp and set ctrlw
   outp(CNTL,ctrlw);

}


//********************************************************************
// FUNCTION NAME: zeroOffset()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Calibrates zero offset error
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//********************************************************************
void a2dClass::zeroOffset(void)
{
   unsigned d=0,i,g2,g10;
   float    sum;
   float    offsetErr[4][4];
   float    bits[4][4];
   unsigned gains10[4] = {1,10,100,100};
   unsigned gains2[4]  = {1, 2,  4,  8};

   clrscr();
   printf("\n\tG10\tG2\t OFFSET\t\t BITS");

   for(g10=0;g10<4;g10++)
      for(g2=0;g2<4;g2++)
         printf("\n\t%d\t%d\t+X.XXXXXX\t+XX.X",g10,g2);

   setRmsOff();
   setAcDc(0);
   setSequencer();
   initTiming(3);
   setChannel(0,0,g10,g2);
   grndInput();
   delay(5);                  //Let new gain values stabilize

   while(!kbhit()){
      for(g10=0;g10<4;g10++){
         for(g2=0;g2<4;g2++){
```

122

```
                setChannel(0,0,g10,g2);
                grndInput();
        lockTrigger();
        resetFifo();
        setFifo();
        unlockTrigger();
        setTrigger();
        delay(1);
        while(getFifoStatus()!=FULL);
        lockTrigger();

        for(i=0,sum=0.0;i<FIFOSIZE;i++){
            d=getFifoData();
            sum+=(float)d*10/2048;
        }
        offsetErr[g10][g2]=((float)(sum/FIFOSIZE)-10)/
                                            (float)(gains10[g10]*gains2[g2]);

        bits[g10][g2]=(float)(offsetErr[g10][g2]*4096/
                                            20*gains10[g10]*gains2[g2]);
            }
            clrscr();
            printf("\n\tG10\tG2\t OFFSET\t\t BITS");
                for(g10=0;g10<4;g10++){
        for(g2=0;g2<4;g2++){
                    printf("\n\t%d\t%d\t%+1.6f\t%+04.1f",g10,g2,
                            offsetErr[g10][g2],bits[g10][g2]);
        }
            }
        }
        freeInput();
        getch();

    }

}

//*****************************************************************
// FUNCTION NAME: grndInput()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Grounds the two diff input for zero adjust
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*****************************************************************
void a2dClass::grndInput(void)
{
    ctrlw |= CAL;
    outpw(CNTL,ctrlw);

}
```

```
//*******************************************************************
// FUNCTION NAME: freeInput()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Ungrounds the two diff inputs
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*******************************************************************
void a2dClass::freeInput(void)
{
    ctrlw &= ~CAL;
    outpw(CNTL,ctrlw);

}


//*******************************************************************
// FUNCTION NAME: zeroAdjust()
// AUTHOR: Randy Walker, based on [MAXUS 95]
// DATE: 27 March 1996
// DESCRIPTION: Adjust the trimmer on the PGA
// RETURNS: void
// CALLS: outpw()
// CALLED BY: none
//*******************************************************************
void a2dClass::zeroAdjust(void)
{
    int       i;
    unsigned d;
    float     sum,offsetErr;

    clrscr();
    printf("\n\nADJUST THE TRIM POT FOR 0.0 OFFSET\n\n");

    setRmsOff();
    setAcDc(0);
    setSequencer();
    initTiming(3);

    while(!kbhit()){
        setChannel(0,0,3,3);
        grndInput();
        lockTrigger();
        resetFifo();
        setFifo();
        unlockTrigger();
        setTrigger();
        while(getFifoStatus()!=FULL);
        lockTrigger();

        for(i=0,sum=0.0;i<FIFOSIZE;i++){
            d=getFifoData();
            sum+=(float)d*10/2048;
        }
```

```c
    offsetErr=((float)(sum/FIFOSIZE)-10)/8000.0;

    printf("\tTHE MEASURED DC OFFSET IS: %+8.6f\r",offsetErr);
    }

    freeInput();
    getch();
}
```

P. A2D.CFG

```
8          ;seqcnt:number_of_seq_addresses_to_load
0          ;mode_sel:__DIFF=1__SE=0
1          ;mode_acdc:_Signal_coupling_select__DC=1__AC=0
8          ;chcnt:_____Number_of_channels_to_sequence_(hex,_1-F)
3125       ;delta_t:__Chan_to_Chan_Sample_rate_in_microsecs_3-8192_=_1/Hz*chcnt
7          ;samprate:__Sample_rate_in_recurrent_mode__0(fast)-7(slow)
0          ;sampindex:_Which_channel_to_sample_in_recurrent_mode
0          0          0          0
1          1          0          0
2          2          0          0
3          3          0          0
4          4          0          0
5          5          0          0
6          6          0          0
7          7          0          0
8          8          0          0
9          A          2          0
A          5          2          0
B          A          2          0
C          5          2          0
D          A          2          0
E          5          2          0
F          A          2          0
```

APPENDIX B: Serial Port Communications Source Code (C++)

A. GLOBALS.H

```
#ifndef __GLOBALS_H
#define __GLOBALS_H

// types
typedef  unsigned char BYTE;
typedef  unsigned short WORD;
typedef  unsigned long  DWORD;

#define  MEM(seg,ofs)      (*((BYTE far*)MK_FP(seg,ofs)))
#define  MEMW(seg,ofs)     (*((WORD far*)MK_FP(seg,ofs)))

enum Boolean   {FALSE, TRUE};

// basic bit twiddles
#define  set(bit)               (1<<bit)
#define  setb(data,bit)        data | set(bit)
#define  clrb(data,bit)        data & !set(bit)
#define  setbit(data,bit)      data = setb(data,bit)
#define  clrbit(data,bit)      data = clrb(data,bit)

// specific to ports
#define  setportbit(reg,bit)   outportb(reg,setb(inportb(reg),bit))
#define  clrportbit(reg,bit)   outportb(reg,clrb(inportb(reg),bit))

#endif
```

B. BUFFER.H

```c
#ifndef __BYTEBUFFER_H
#define __BYTEBUFFER_H

#include "toetypes.h"

#define  BYTEBUFSIZE     32

/************************************************************************
CLASS:Buffer
AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Base class for use as a polymorphic reference in the serial port code
   which defines a buffer to be used in serial port communications.
*************************************************************************/

class Buffer  {

   public:

      //Constructor
      Buffer(WORD sz) : getPtr(0), putPtr(0), size(sz)  {}

      //Checks for the arrival of new characters in the buffer
      virtual Boolean  hasData(){ return Boolean(putPtr !=
                                 getPtr); }

      //How much of the Buffer is used (rounded percentage
      //0 - 100)
      virtual int       capacityUsed();
      //Read from the buffer
      virtual Boolean   Get(BYTE&) = 0;
      //Read to the buffer
      virtual void      Add(BYTE) = 0;

   protected:

      //Increment the pointer to next position
      void  inc(WORD& index) { if (++index == size) index = 0; }

         //Decrement the pointer
      WORD before(WORD index){ return ((index == 0) ? size - 1 :
                               index - 1);}

      WORD      getPtr,  //Location of unread data
                putPtr,  //Location to read data to
                size;    //Size of the buffer in bytes
};
```

128

```
/****************************************************************
   Defines a single buffer of a specified size for buffering charaters
   received via serial port.
****************************************************************/

class byteBuffer : protected Buffer {

   public:

      byteBuffer(BYTE sz=BYTEBUFSIZE);
      ~byteBuffer()      { delete [] buf; }

      Buffer::hasData;

      Buffer::capacityUsed;

      // buffer extraction
      Boolean      Get(BYTE&);

      // buffer insertion
      void      Add(BYTE ch);
      void      Add(const char*);
      byteBuffer&  operator += (BYTE ch){ Add(ch); return *this;
                                          }

   protected:

      BYTE*      buf;
};
#endif
```

C. BUFFER.CPP

```
#include <iostream.h>
#include <stdio.h>

#include "globals.h"
#include "buffer.h"

//*********************************************
// Buffer
//*********************************************
// Returns the percentage of the buffer in use
int
Buffer::capacityUsed()
{
   int cap = (putPtr + size) % size - getPtr;
   return 100 * cap / size;
}
```

```cpp
//*********************************************
// byteBuffer
//*********************************************
//Constructor, instantiates a buffer
byteBuffer::byteBuffer(BYTE sz) : Buffer(sz)
{
   buf = new BYTE[size];
}

//Reads a charcter from the buffer
Boolean
byteBuffer::Get(BYTE& data)
{
   if (hasData())  {
      data = buf[getPtr];
      inc(getPtr);
   return TRUE;
   }
   return FALSE;
}

//Writes a character to the buffer and checks for buffer overflow
void
byteBuffer::Add(BYTE ch)
{
   buf[putPtr] = ch;
   inc(putPtr);
   if (!hasData())  {   // if there's no data after adding data,
                        // it overflowed
      cerr << "\nError: byteBuffer overflow\n";
   }
}

//Writes a character to the buffer
void
byteBuffer::Add(const char* s)
{
   while (*s)
      Add(*s++);
}
```

130

D. GPSBUFF.H

```
ifndef __GPSBUFF_H
#define __GPSBUFF_H

#include "buffer.h"
#include "toetypes.h"

#define   GPSBLOCKS      4
#define   LINE_FEED      10
#define   CARR_RETURN    13


/****************************************************************
   Class buffers GPS position messages via serial port communications.
   Uses a multiple buffer system in which each buffer is capable of holding a
single position message. Buffers are filled and processed sequentially in a
round robin fashion. Messages are checked for validity only upon attempted
reads from the buffer.
****************************************************************/

class GPSbuffer : public Buffer  {

public:

   GPSbuffer(BYTE GPSblocks=GPSBLOCKS);
   ~GPSbuffer(){delete [] block;}

   Boolean   hasData();       // a complete structure is ready
   Boolean   Get(BYTE&)    {return FALSE;}
   Boolean   Get(GPSdata);  // get a complete structure filled in
   void      Add(BYTE ch);  // build the structure as each byte
                            // is added

protected:

   Boolean   validHeader(GPSdata); // check a block for valid
                                   // header
   GPSdata   *block;       // hold the buffered GPS data
   WORD      current,      // the current GPS block in use
             last;          // the last GPS block in use

   BYTE      *putPlace;  // place to put the next charater received
};

#endif
```

131

E. GPSBUFF.CPP

```cpp
#include <iostream.h>
#include <stdio.h>

#include "gpsbuff.h"

/****************************************************************
PROGRAM:GPSbuffer (Constructor)
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Allocates message buffers, indicate that no data has
  beenreceived by equalizing current and last and set position
   into which initial character will be read.
RETURNS:nothing.
CALLED BY:navigator class (nav.h)
CALLS:none.
****************************************************************/


GPSbuffer::GPSbuffer(BYTE GPSblocks) :
   current(0), last(0),
   Buffer(GPSblocks) // Call to base class constructor
{
   block = new GPSdata[GPSblocks];  //Create an array of GPSdata
                                     //elements
   putPlace = &(block[current][0]); //Set the place for the
                                     // first character
}
```

```
/****************************************************************
PROGRAM:GPSbuffer::Add
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Interrupt driven routine which writes incoming characters into the
gps buffers
RETURNS:nothing.
CALLED BY:  interupt driven by bufferedSerialPort
CALLS:none.
****************************************************************/

void
GPSbuffer::Add(BYTE data){

    static BYTE lastChar(data);  //Holds last for <cr> <lf> detection
    static Boolean lfFlag = FALSE; //True when message end is detected

    //Is a new message starting?
    if (lfFlag && (data == '@')) {
       last = current;  // Set last to buffer with newest message.
       inc(current);    // Set current to the next buffer
       // Set putPlace to the beginning of the next buffer.
       putPlace = &(block[current][0]);
       lfFlag = FALSE;  // reset for end of next message.
    }

    *putPlace++ = data;// Write character into the buffer.

    //Has the end of a message been received?
    if ((lastChar == CARR_RETURN) &&
       (data == LINE_FEED)) {
       lfFlag = TRUE;
    }
    lastChar = data; //Save last character for <cr> <lf> detection

}
```

```
/*****************************************************************
PROGRAM:GPSbuffer::Get
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Checks to see if a
    new message has arrived, copies it into the
    input argument data and returns a flag to indicate whether a
    newmessage was received.
RETURNS:TRUE, if a new valid position has been received.
         FALSE, otherwise
CALLED BY:navPosit (nav.cpp)
          initializeNavigator (nav.cpp)
CALLS:GPSbuffer::hasData
*****************************************************************/

Boolean
GPSbuffer::Get(GPSdata data)
{
    // Has a new valid message been received.
    if (hasData())  {
       // Copy the message out of the buffer.
       memcpy (data, block + last, GPSBLOCKSIZE);
       // Indicate that this message has been read.
       last = current;
       return TRUE;
    }
    else {
       return FALSE;
    }
}
```

```
/**************************************************************
PROGRAM:GPSbuffer::hasData
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Determines whether a new message has been received and
   checks to see if it has a valid header.
RETURNS:TRUE, if a new valid message has been received.
CALLED BY:   GPSbuffer::Get (buffer.cpp)
CALLS:       validHeader (buffer.cpp)
**************************************************************/

Boolean
GPSbuffer::hasData()
{

    // Has a new message with a valid header been received
    if (last != current) {
        if (validHeader(block[last])) {
            return TRUE;
        }
        else {
            return FALSE;
        }
    }
    return FALSE;
}

/**************************************************************
PROGRAM:validHeader
AUTHOR:Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Checks to see if a message has the proper header for a
   Motorola position message. (@@Ea)
RETURNS:TRUE, if the header is valid. FALSE, otherwise.
CALLED BY:GPSbuffer::hasData (buffer.cpp)
CALLS:none.
COMMENTS:
**************************************************************/

Boolean
GPSbuffer::validHeader(GPSdata dataPtr)
{

    if ((dataPtr[0] == '@') && (dataPtr[1] == '@') &&
                              (dataPtr[2] == 'E') &&
                               (dataPtr[3] == 'a')) {
        return TRUE;
    }
    else {
        return FALSE;
    }
}
```

F. PORTBANK.H

```
#ifndef __PORTBANK_H
#define __PORTBANK_H

#include "serial.h"
#include "buffer.h"

/**************************************************************
CLASS:portBank
AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
DATE:11 July 1995
FUNCTION:Manages up to four bufferdSerialPort instances.
**************************************************************/

class  portBank  {

   public:

      portBank();

      ~portBank()    { cleanup(); }

      bufferedSerialPort&  Init(COMport portnum, BYTE irq,
       BaudRate,ParityType,BYTE wordlen, BYTE stopbits,
       handShake, Buffer&);

      void  cleanup();

      friend IntHandlerType COM1handler, COM2handler,
       COM3handler,COM4handler;

   protected:

      bufferedSerialPort*  ports[4];

};

extern portBank    COMports;

#endif
```

G. PORTBANK.CPP

```cpp
#include <iostream.h>

#include "serial.h"
#include "buffer.h"
#include "portbank.h"

portBank    COMports;

// Constructor, sets up array of ports
portBank::portBank()
{
    for (int i = 0; i < 4; i++)
        ports[i] = 0;
}

// Resets all ports to the original parameters
void
portBank::cleanup()
{
    for (int i=0; i<4; i++)
        delete ports[i];
}

// Initializes a serial port based up the input arguments
bufferedSerialPort&
portBank::Init(COMport portnum, BYTE irq, BaudRate baud, ParityType parity,
      BYTE wordlen, BYTE stopbits, handShake shake, Buffer& buf)
{
    int    index = BYTE(portnum) - 1;
    if (ports[index])
        delete ports[index];
    ports[index] = new bufferedSerialPort(portnum, irq, baud,
                        parity,wordlen, stopbits, shake, buf);
    return *ports[index];
}

// Three specific interrupt handlers which map each interupt to ///the
proper ISR.
void
interrupt
COM1handler(...)
{
    COMports.ports[0]->processInterrupt();
    EOI;
}

void
interrupt
COM2handler(...)
{
    COMports.ports[1]->processInterrupt();
    EOI;
```

```
}

void
interrupt
COM3handler(...)
{
    COMports.ports[2]->processInterrupt();
    EOI;
}

void
interrupt
COM4handler(...)
{
    COMports.ports[3]->processInterrupt();
    EOI;
}
```

H. SERIAL.H

```
#ifndef __SERIAL_H
#define __SERIAL_H

#include <dos.h>
#include <stdio.h>
#include "globals.h"
#include "buffer.h"

// user defines
#define     ALMOST_FULL     80      // % full to turn off DTR

// leave the following alone - hardware specific

enum   COMport          {COM1=1, COM2, COM3, COM4};
enum   BaudRate         {b300, b1200, b2400, b4800, b9600};
enum   ParityType       {ERROR=-1, NOPARITY, ODD, EVEN};
enum   handShake        {NONE, RTS_CTS, XON_XOFF};
enum   Shake            {off, on};
enum   interruptType    {rx_rdy, tx_rdy, line_stat, modem_stat};

#define   BIOSMEMSEG   0x40
#define   DLAB         0x80
#define   IRQPORT      0x21
#define   EOI          outportb(0x20,0x20)

#define   COM1base     MEMW(BIOSMEMSEG,0)
#define   COM2base     MEMW(BIOSMEMSEG,2)
#define   COM3base     0x03e8
#define   COM4base     0x02e8

#define   TX        (portBase)
#define   RX        (portBase)
#define   IER       (portBase+1)
```

```
#define  IIR      (portBase+2)
#define  LCR      (portBase+3)
#define  MCR      (portBase+4)
#define  LSR      (portBase+5)
#define  MSR      (portBase+6)
#define  LO_LATCH (portBase)
#define  HI_LATCH (portBase+1)

/**************************************************************
   CLASS:serialPort
   AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:11 July 1995
   FUNCTION:  Defines a simple serial port.
***************************************************************/

class serialPort  {

   public:

      serialPort(COMport port, BaudRate, ParityType, BYTE wordlen,
           BYTE stopbits, handShake);

      Boolean    Send(BYTE data);
      Boolean    Get(BYTE& data);

      inline Boolean dataReady();
      Boolean statusChanged()
          { return Boolean((ifportbit(MSR,0) || ifportbit(MSR,1))); }

      // the rest are only if handshake is specified as RTS_CTS
      Boolean    isCTSon()              { return ifportbit(MSR,4); }
      Boolean    isDSRon()              { return ifportbit(MSR,5); }

      void       setDTRon()             { setportbit(MCR,0); }
      void       setDTRoff()            { clrportbit(MCR,0); }
      void       toggleDTR();

      void       setRTSon()             { setportbit(MCR,1); }
      void       setRTSoff()            { clrportbit(MCR,1); }
      void       toggleRTS();

   protected:

      WORD           portBase;
      handShake      ShakeType;
      Shake          DTRstate,
                       RTSstate;

      inline Boolean   ifportbit(WORD, BYTE);
      inline void      toggle(Shake&);

};
```

139

```
// this is the type for a standard interrupt handler
typedef void interrupt (IntHandlerType)(...);

/*****************************************************************
   CLASS:bufferedSerialPort
   AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:11 July 1995
  FUNCTION:  Defines a buffered serial port which is interrupt driven
   on receive, and buffers all incoming characters in the specified buffer
*****************************************************************/

class bufferedSerialPort : public serialPort  {

   public:

      bufferedSerialPort(COMport portnum, BYTE irq, BaudRate,
       ParityType, BYTE wordlen, BYTE stopbits, handShake, Buffer& );

      ~bufferedSerialPort();

      Boolean     Get(BYTE& data);      // buffered version

   protected:

      Buffer&     buf;

      BYTE irqbit,//Value to allow enable PIC interrupts for COM port
         origirq,//keep the original value of the 8259 mask register
         comint;

      void  processInterrupt();     // buffers the incoming character

      IntHandlerType *origcomint;// keep the original vector for
                                 //restoring later

      // this allows the actual handlers to access processInterrupt()
      friend IntHandlerType COM1handler, COM2handler, COM3handler,
                        COM4handler;

};

#endif
```

I. SERIAL.CPP

```cpp
#include <iostream.h>
#include <stdio.h>
#include "serial.h"

/******************************************************************
      Usage Note: Because of the interrupt handlers used, you MUST call your
bufferedSerialPort objects port1, port2, or port3, so the
right handler gets called and can properly service the interrupt.
******************************************************************/

/******************************************************************
    PROGRAM: serialPort (Constructor)
    AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
    DATE:11 July 1995
    FUNCTION:
    Initializes the one of the Serial Ports.
        1) Determines the base I/O port address for the given COM port
        2) Sets the 8259 IRQ mask value
        3) Initializes the port parameters - baud, parity, etc.
        4) Calls the routine to initialize interrupt handling
        5) Enables DTR and RTS, indicating ready to go
******************************************************************/

serialPort::serialPort(COMport port, BaudRate speed, ParityType parity,BYTE
wordlen, BYTE stopbits, handShake hs) :
    DTRstate(off), RTSstate(off), ShakeType(hs)
{
    switch (port)  {
      case COM1:  portBase = COM1base;
                break;
      case COM2:  portBase = COM2base;
                break;
      case COM3:  portBase = COM3base;
                break;
      case COM4:  portBase = COM4base;
                break;
    }  //switch

    const WORD     bauddiv[] = {0x180, 0x60, 0x30, 0x18, 0xC};
     // Change 1
    outportb(IER,0);          // disable UART interrupts
    (void)inportb(LSR);
    (void)inportb(MSR);
    (void)inportb(IIR);
    (void)inportb(RX);
    outportb(LCR,DLAB);//set DLAB so can set baud rate(read only
                       // port)
    outportb(LO_LATCH,bauddiv[speed] & 0xFF);
    outportb(HI_LATCH,(bauddiv[speed] & 0xFF00) >> 8);
    setportbit(MCR,3);                    //turn OUT2 on

    BYTE  opt = 0;
```

```
    if (parity != NOPARITY)  {
      setbit(opt,3);          // enable parity
       if (parity == EVEN)  // set even parity bit. if odd, leave bit 0
         setbit(opt,4);
    }
     // now set the word length. len of 5 sets both bits 0 and 1 to
     // 0, 6 sets to 01, 7 to 10 and 8 to 11
    opt |= wordlen-5;
    opt |= --stopbits << 2;
    outportb(LCR,opt);

    if (ShakeType == RTS_CTS)  {
      setDTRon();
      setRTSon();
    }
}


/*************************************************************
   PROGRAM: Get
   AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:11 July 1995
   FUNCTION:Gets a byte from the port. Returns true if there's one there, and
fills in the byte parameter. If theres no character, the parameter is left alone,
and false is returned
**************************************************************/

Boolean
serialPort::Get(BYTE& data)
{
    if (dataReady())  {         //make sure there's a char there
      data = inportb(RX);       //read character from 8250
      return TRUE;
    }
    else
      return FALSE;
}
```

142

```
/****************************************************************
    PROGRAM: Send
    AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
    DATE:11 July 1995
    FUNCTION:Outputs a single character to the port. Returns Boolean
    status indicating whether successful
****************************************************************/

Boolean
serialPort::Send(BYTE data)
{
    while (!(ifportbit(LSR,5)))        // wait until THR ready
      ;  // NULL statement

    switch (ShakeType)  {            .
       case NONE:
         outportb(TX,data);
          return TRUE;
      case RTS_CTS:
         if (isCTSon() && isDSRon())  {
             outportb(TX,data);
             return TRUE;
          }
          else  return FALSE;
      case XON_XOFF:
      default:
      // add this later if needed
       break;
    }
    return FALSE;
}
```

```
/****************************************************************
    PROGRAM: dataReady
    AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
    DATE:11 July 1995
    FUNCTION:Checks port to see if a character has arrived.
****************************************************************/

inline
Boolean
serialPort::dataReady()
{
/*
    if (ifportbit(LSR,1)) {
      cerr <<"\nOverrun Error\n";
    }
    if (ifportbit(LSR,2)) {
      cerr <<"\nParity Error\n";
    }
    if (ifportbit(LSR,3)) {
      cerr <<"\nFraming Error\n";
    }
*/
    return ifportbit(LSR,0);
}

/****************************************************************
    PROGRAM: ifportbit
    AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
    DATE:11 July 1995
    FUNCTION:
****************************************************************/

inline
Boolean
serialPort::ifportbit(WORD reg, BYTE bit)
{
    BYTE on = inportb(reg);
    on &= set(bit);
    return Boolean(on == set(bit));
}
```

144

```
/*****************************************************************
   PROGRAM: toggleDTR
   AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:11 July 1995
   FUNCTION: toggles Data Transmit Ready if RTS_CTS is off
*****************************************************************/

void
serialPort::toggleDTR()
{
   if (ShakeType != RTS_CTS)
     return;
   if (DTRstate == off)
     setDTRon();
   else
     setDTRoff();
   toggle(DTRstate);
}

/*****************************************************************
   PROGRAM: toggleRTS
   AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:11 July 1995
   FUNCTION: toggle Ready to Send (RTS) if RTS_CTS is on.
*****************************************************************/

void
serialPort::toggleRTS()
{
   if (ShakeType != RTS_CTS)
     return;
   if (RTSstate == off)
     setRTSon();
   else
     setRTSoff();
   toggle(RTSstate);
}

/*****************************************************************
   PROGRAM: toggle
   AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:11 July 1995
   FUNCTION: toggles value of the input variable
*****************************************************************/

inline
void
serialPort::toggle(Shake& h)
{
   if (h == off)
     h = on;
   else
     h = off;
}
```

```
//************************************************************
//     bufferedSerialPort
//************************************************************

/************************************************************
   PROGRAM:bufferedSerialPort (Constructor)
  AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:11 July 1995
   FUNCTION:
    Initializes the interrupts for the Serial Port.
            1) takes over the original COM interrupt
            2) sets the port bits, parity, and stop bit
            3) enables interrupts on the 8250 (async chip)
            4) enables the async interrupt on the 8259 PIC
************************************************************/

bufferedSerialPort::bufferedSerialPort(COMport portnum, BYTE irq,
                       BaudRate baud, ParityType parity, BYTE wordlen,
                       BYTE stopbits, handShake hs, Buffer& b)
    : serialPort(portnum, baud, parity, wordlen, stopbits, hs),
     buf(b), irqbit(irq), comint(irqbit+8)
{
    if (ShakeType == RTS_CTS)  { // turn it off first, because it
                                 // was enabled
      setDTRoff();               // in the base class
      setRTSoff();
    }
    origcomint = getvect(comint); //remember the original vector

    switch (portnum) {
     case COM1:
       setvect(comint,COM1handler); //point to the new handler
       break;
     case COM2:
       setvect(comint,COM2handler); //point to the new handler
       break;
     case COM3:
       setvect(comint,COM3handler); //point to the new handler
       break;
     case COM4:
       setvect(comint,COM4handler); //point to the new handler
       break;
    }

//   setportbit(MCR,3);                  //turn OUT2 on
    disable();        // disable all interrupts - critical section
    setportbit(IER,rx_rdy);             //enable ints on receive only
    origirq = inportb(IRQPORT);         //remember how it was
    clrportbit(IRQPORT,irqbit);         //enable COM ints

    if (ShakeType == RTS_CTS)  {
      setDTRon();
      setRTSon();
    }
```

146

```
    enable();
    EOI;
}


/****************************************************************
    PROGRAM: ~bufferedSerialPort
    AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
    DATE:11 July 1995
    FUNCTION:
     Resets the interrupts.  1) disables the 8250 (async chip)
                             2) disables the interrupt chip for async int
                             3) resets the 8259 PIC

****************************************************************/

bufferedSerialPort::~bufferedSerialPort()
{
    setvect(comint,origcomint);     //set the interrupt vector back
    outportb(IER,0);                //disable further UART interrupts
    outportb(MCR,0);                //turn everything off
    outportb(IRQPORT,origirq);
    EOI;
}


/****************************************************************
    PROGRAM: Get
    AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
    DATE:11 July 1995
    FUNCTION:  Calls Get base on buffer type
****************************************************************/

Boolean
bufferedSerialPort::Get(BYTE& data)
{
    return buf.Get(data);
}


/****************************************************************
    PROGRAM: processInterupt
    AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
    DATE:11 July 1995
    FUNCTION:  Calls the ISR based upon buffer type
****************************************************************/

void
bufferedSerialPort::processInterrupt()
{
    if (dataReady())  {          //make sure there's a char there
      BYTE data = inportb(RX);    //read character from 8250
      buf.Add(data);
      if (ShakeType == RTS_CTS && buf.capacityUsed() > ALMOST_FULL)
        setDTRoff();
    }
}
```

147

```
/*****************************************************************
   PROGRAM: showPorts
   AUTHOR:Frank Kelbe, Eric Bachmann, Dave Gay
   DATE:11 July 1995
   FUNCTION:   Prints interupt vector addresses
*****************************************************************/

int
showPorts()
{
   BYTE* p = (BYTE*)COM2base;
     p += 5;
     fprintf(stderr,"%X    ",*p++);
     fprintf(stderr,"%X\n",*p++);
   fprintf(stderr,"IRQPORT = %X", inportb(IRQPORT));
   return 0;
}
```

J. COMPBUFF.H

```
#ifndef __COMPBUFF_H
#define __COMPBUFF_H

#include "buffer.h"
#include "toetypes.h"

#define   COMPBLOCKS      8
#define   LINE_FEED       10
#define   CARR_RETURN     13


/*************************************************************************

  Class buffers COMPASS messages received via serial port communications.
  Uses a multiple buffer system in which each buffer is capable of
  holding a single message. Buffers are filled and processed
  sequentially in a round robin fashion. Messages are checked for validity
  only upon attempted reads from the buffer.

*************************************************************************/

class compBuffer : public Buffer  {

public:

    compBuffer(BYTE compBlocks=COMPBLOCKS);
    ~compBuffer(){delete [] block;}

    Boolean  hasData();       // a complete structure is ready
    Boolean  Get(BYTE&)    {return FALSE;} //Satisfy inheritence requirements
    Boolean  Get(compData);  // get a complete structure filled in
    void     Add(BYTE ch);  // build the structure as each byte is added

protected:

    Boolean  validHeader(compData); // check a block for valid header
    compData *block;         // Pointer to array of compass messages
    WORD     current,        // the current comp block in use
             last;           // the last comp block in use

    BYTE     *putPlace;  // place to put the next charater received
};

#endif
```

K. COMPBUFF.CPP

```cpp
#include <iostream.h>
#include <stdio.h>

#include "compbuff.h"

/**********************************************************************
    PROGRAM:compBuffer (Constructor)
    AUTHOR:Eric Bachmann, Randy Walker
    DATE:28 April 1996
    FUNCTION:
    Allocates message buffers, indicates that no data has been
    received by equalizing current and last and sets the position
    into which initial character will be read.
    RETURNS:nothing.
    CALLED BY:compass class (compass.h)
    CALLS:none.
***********************************************************************/

compBuffer::compBuffer(BYTE compBlocks):
    current(0), last(0),
    Buffer(compBlocks) //Call to constructor of the base class
{
    block = new compData[compBlocks]; // Create an array of message buffers
    putPlace = &(block[current][0]); // Set position for first character
}
```

```
/********************************************************************
   PROGRAM:compBuffer::Add
   AUTHOR:Eric Bachmann, Randy Walker
   DATE:28 April 1996
   FUNCTION:
   Interrupt driven routine which writes incoming characters
   into the coompass message buffers
   RETURNS:nothing.
   CALLED BY:  interupt driven by bufferedSerialPort
   CALLS:none.
********************************************************************/

void
compBuffer::Add(BYTE data){

   static Boolean lfFlag = FALSE; //True, if message end detected
   static int messageCount(0); // Counts characters in current message

   //Is a new message starting?
   if (lfFlag && (data == '$')) {

    last = current;  // Set last to buffer with newest message.
    inc(current);    // Set current to the next buffer

    // Set putPlace to the beginning of the next buffer.
    putPlace = &(block[current][0]);
    lfFlag = FALSE;  // reset for end of next message.
   }

   *putPlace++ = data;// Write character into the buffer.
   messageCount++;

   //Has the end of a message been received (<cr><lf>)?
   if (data == LINE_FEED) {
    lfFlag = TRUE;
   }
}
```

151

```
/*********************************************************************
    PROGRAM:compBuffer::Get
    AUTHOR:Eric Bachmann, Randy Walker
    DATE:28 April 1996
    FUNCTION:
    Checks to see if a new message has arrived, copies it into the
    input argument data and returns a flag to indicate whether a new
       message was received.
    RETURNS:TRUE, if a new valid position has been received.
       FALSE, otherwise
    CALLED BY: compass.cpp
    CALLS:compBuffer::hasData

*********************************************************************/

Boolean
compBuffer::Get(GPSdata data)
{
    // Has a new valid message been received.
    if (hasData())   {
         // Copy the message out of the buffer.
      memcpy (data, block + last, COMPSIZE);
         // Indicate that this message has been read.
      last = current;
      return TRUE;
    }
    else {
      return FALSE;
    }
}
```

```
/******************************************************************
   PROGRAM:compBuffer::hasData
   AUTHOR:Eric Bachmann, Randy Walker
   DATE:28 April 1996
   FUNCTION:
   Determines whether a new message has been received and checks
      to see if it has a valid header.
   RETURNS:TRUE, if a new valid message has been received.
   CALLED BY:   compBuffer::Get
   CALLS:       validHeader (compBuffer.cpp)
******************************************************************/

Boolean
compBuffer::hasData()
{
   if ((last != current) && (validHeader(block[last]))) {
         return TRUE;
   }
   else {
         return FALSE;
   }
}
/******************************************************************
   PROGRAM:validHeader
   AUTHOR:Eric Bachmann, Randy Walker
   DATE:15 May 1996
   FUNCTION:
   Checks to see if a message has the proper header for a compass
   message. ($C)
   RETURNS:TRUE, if the header is valid. FALSE, otherwise.
   CALLED BY:compBuffer::hasData
   CALLS:none.
   COMMENTS:
******************************************************************/

Boolean
compBuffer::validHeader(compData dataPtr)
{
  if ((dataPtr[0] == '$') &&
      (dataPtr[1] == 'C')) {
   return TRUE;
  }
  else {
   return FALSE;
  }
}
```

153

LIST OF REFERENCES

Bachmann, E.R. and Gay, D., "Design and Evaluation of an Integrated GPS/INS System for Shallow-water AUV Navigation (SANS)," Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1995.

Bachmann, E.R., McGhee, R.B., Whalen, R.H., Steven, R., Walker, R.G., Clynch, J.R., Healey, A.J., and Yun, X.P., "Evaluation of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)," *Proceedings of the Symposium on Autonomous Underwater Vehicle Technology*, Monterey, California, June, 1996.

Bergem, O., "A Multibeam Sonar Based Positioning System for an AUV," *Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology*, Durham, New Hampshire, September 27-29 1993, pp. 291-299.

Bowditch, N., *American Practical Navigator, Vol. 1 and 2*, Defense Mapping Agency Hydrographic/Topographic Center, 1984.

Brown, R.G and Hwang, P.Y.C., *Introduction to Random Signals and Applied Kalman Filters*, 2nd Edition, John Wiley and Sons, New York, 1992.

Brutzman, D.P., Burns, M., Campbell, M., Davis, D.T., Healey, A.J., Holden, M., Leonhardt, B., Marco, D., McClarin, D., McGhee, R.B. and Whalen, R.,"NPS Pheonix AUV Software Integration and In-Water Testing," *Proceedings of the Symposium on Autonomous Underwater Vehicle Technology*, Monterey, California, June, 1996.

Cox, I.J. and Wilfong, G.T., *Autonomous Robot Vehicles*, Springer-Verlag, New York, 1990.

DATEL BWR Series Data Sheet, DATEL, Inc., September, 1993.

Frequency Devices DP74 Series Data Sheet, Frequency Devices, January, 1996.

Grose, B.L., "The Application of the Correlation Sonar to Autonomous Underwater Vehicle Navigation," *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology*, IEEE Oceanic Engineering Society, June 2-3 1992, Washington, D.C., pp. 298-303.

Healey, A.J. "Evaluation of the NPS PHOENIX Autonomous Underwater Vehicle Hybrid Control System," *Proceedings of ACC'95 Conference*, Seattle, Washington, June, 1995.

Healey, A.J. and Lienard, D., "Multivariable Sliding Mode Control for Autonomous Diving and Steering of Unmanned Underwater Vehicles," *IEEE Journal of Oceanic Engineering*, vol. 18 no. 3, July, 1993.

Kwak, S.H., Stevens, C.D., Clynch, J.R., McGhee, R.B., and Whalen, R.H., "An Experimental Investigation of GPS/INS integration for Small AUV Navigation," *Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology (UUST)*, September 27-29 1993, Durham, New Hampshire, pp. 239-251.

Leu, C.T., Chao, J.J., and Lee, T.S., "GPS Based Underwater Positioning - A System Design," *Proceedings of The Institute of Navigation GPS-93*, Salt Lake City, Utah, September 22-24 1993, pp. 745-754.

Logsdon, T., *The Navstar Global Positioning System*, Van Nostrand Reinhold, New York, 1992.

Matthews, M.B., "A Description of the Hardware and Software Interface to the Systron-Donner MotionPak Inertial Sensor Unit for ROV Tiberon," Monterey Bay Aquarium Research Institute Internal Correspondence, Monterey Bay Aquarium Research Institute, March 6, 1995.

MAXUS E.S.P 386sx/486slc Scamp II User's Manual, Maxus Electronics Corp., July, 1995.

McGhee, R.B., Clynch, J.R., Healey, S.H., Kwak, S.H., Brutzman, D.P., Yun, X.P., Norton, N.A., Whalen, R.H., Bachmann, E.R., Gay, D.L. and Schubert, W.R., "An Experimental Study of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)," *Proceedings of the Ninth International Symposium on Unmanned Untethered Submersible Technology (UUST)*, September 25-27 1995, Durham, New Hampshire.

McKeon, J.B., *Integration of GPS into a Small Underwater Navigation System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1992.

Nagengast, S., *Correction of Inertial Measurements Using GPS Update for Underwater Navigation*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1992, pp. 4-6.

Norton, N.A., Evaluation of Hardware and Software for a Small Autonomous Underwater Vehicle Navigation System (SANS), Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1994.

Oncore User's Guide, Motoraola Inc., August 1995.

Parkinson, B.W., "Overview," *Global Positioning System*, Vol. 1, The Institute of Navigation, Washington, D.C., 1980, pp. 1-2.

Schubert, W.R. and Whalen, R.H., "Design and Implementation of an Integrated GPS/INS System for Shallow-Water AUV Navigation," *Technical Report No. CS-95-003*, Naval Postgraduate School, Monterey, California 93943, July, 1995.

Souen, K., and Nishida, T., "The World's Smallest 8-Channel GPS Receiver," *Proceedings of The Institute of Navigation GPS-92*, Albuquerque, New Mexico, September 16-18 1992, pp. 707-713.

Systron-Donner Model MP-GCCCQAAB MotionPaK IMU, Systron-Donner, Inc., Concord, California.

TCM2 Electronic Compass Module User's Manual, Precision Navigation, Inc., June, 1995.

Tuohy, S.T., Patrikalakis, N.M., Leonard, J.J., Bellingham, J.G., and Chryssostomidis, C., "AUV Navigation Using Geophysical Maps with Uncertainty," *Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology (UUST)*, Durham, New Hampshire, September 27-29 1993, pp. 265-276.

Wooden, W. H., "NAVSTAR Global Positioning System: 1985," *Proceedings of the First International Symposium on Precise Positioning with the Global Positioning System*, April 1985, pp. 23-32.

Youngberg, J.W., "A Novel Method for Extending GPS to Underwater Applications," *NAVIGATION, Journal of The Institute of Navigation*, vol. 38, no. 3, Fall 1991.

Yuh, J., *Underwater Robotic Vehicles: Design and Control*, TSI Press, Albuquerque, New Mexico, 1995.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA   22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA   93943-5101

3. Director, Marine Corps Research Center 2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, VA 22134-5107

4. Director, Studies and Analysis Division 1
   MCCDC, Code C45
   3300 Russell Road
   Quantico, VA 22134-5130

5. Director, Training and Education 1
   MCCDC, Code C46
   1019 Elliot Rd.
   Quantico, VA 22134-5027

6. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA   93943-5121

7. Chairman, Code CS 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA   93943-5118

8. Prof. Xiaoping Yun, , Code EC/Yx 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA   93943-5121

9. Prof.Robert B. McGhee, Code CS/Mz   1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA   93943-5118

10. Dr. James Clynch, Code OC/CL   1
    Department of Oceanography
    Naval Postgraduate School
    Monterey, CA   93943-5122

11. Russ Whalen, Code CS   1
    Department of Computer Science
    Naval Postgraduate School
    Monterey, CA   93943-5118

12. CMDR R.W. Olson, USNR   4
    308 N.E. 3rd St.
    Little Falls, MN 56345

13. LT. Eric R. Bachmann, USN   1
    1281 Spruance Rd.
    Monterey, CA 93940

14. Mr. Norman Caplan   1
    National Science Foundation
    BES, Room 565
    4201 Wilson Blvd.
    Arlington, VA 22230

15. Michael B. Matthews   1
    Monterey Bay Aquarium Research Institute
    P.O. Box 628
    Moss Landing, CA 95039